VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Podpůrný systém pro správu úkolů v rámci projektu Virtlab

Support System for Project Management in Virtlab Project

2010 Roman Sklenář

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.

V Ostravě 7. května 2010 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Rád bych na tomto místě poděkoval Ing. Adamu Janoškovi za odborné vedení během mé práce.

Abstrakt

Tato bakalářská práce se zabývá tvorbou informačního systému pro správu projektů a úkolů. V úvodu práce bude postupně rozebrána problematika projektového řízení a vybraná metodika organizace práce. Dalším krokem bude návrh informačního systému, který bude navrhnut s ohledem na všechny aspekty problematiky pokryté v úvodu práce. Jedním z požadavků je, aby informační systém také podporoval vybranou metodiku organizace práce, k čemuž je také v analýze přihlédnuto. V analýze také provedu výběr vhodného aplikačního frameworku pro implementaci informačního systému. Závěrečná část práce se pak zabývá částečnou implementací navržené aplikace a navrhuje její další možná rozšíření.

Klíčová slova: Informační systém, webová aplikace, Ruby, Ruby on Rails, osobní produktivita, řízení projektů, Mít vše hotovo

Abstract

This bachelor thesis deals with creation of an information system for managing projects and tasks. First and second chapter of thesis discusses and covers issues of successful project management and choosen methodology of work organization. The next step will be to design an information system that will be designed with regard to all aspects of the issues covered in the introduction. One requirement is that the information system will also support the selected method of work organization, which is also taken into account in the analysis. The analysis also make a choice of application framework for the implementation of information system. The final part deals with the partial implementation of the proposed applications and suggests a possible further extensions.

Keywords: Information system, web applications, Ruby, Ruby on Rails, personal productivity, project management, Getting Things Done

Seznam použitých zkratek a symbolů

AJAX	–	Asynchronous JavaScript and XML
CSRF	–	Cross-site Request Forgery
DDL	–	Data Definition Language
ERP	–	Enterprise Resource Planning
GTD	–	Getting Things Done
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
IDE	–	Integrated development enviroment
MVC	–	Model-view-controller
ORM	–	Object-relational mapping
PPM	–	Project and Portfolio Management
REST	–	Representational State Transfer
SŘBD	–	Systém řízení báze dat
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
TDD	–	Test-driven Development
UI	–	User interface
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
XSS	–	Cross-site Scripting

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Webové aplikace zaznamenaly za poslední desetiletí nezanedbatelný růst. Z původně jednoúčelových záležitostí pro konzumaci obsahu se stávají platformou k dynamické tvorbě online obsahu a umožňují s ním i interaktivní práci. Původně okrajové a spíše experimentální technologie, standardy a frameworky časem dozrály, ukazují se v praxi jako velmi efektivní a nalézají si cestu do hlavního proudu a určují tak směr dalšího vývoje. Pomalu se začínají objevovat webové aplikace, ze kterých se staly vyspělé systémy, které dokonce občas nemají důstojnou desktopovou alternativu. Serverové technologie se tak stávají jednou z nejdynamičtěji se rozvíjejících odvětví softwarového průmyslu.

Jak už název této bakalářské práce napovídá, jejím cílem je najít a implementovat vhodné řešení pro správu úkolů týmu vývojářů v rámci projektu Virtlab. Implementace bude provedena formou webové aplikace. Dalším z cílů bakalářské práce je zmapovat a pokrýt problematiku projektového řízení, nastudovat metodiku organizace práce Getting Things Done a prozkoumat její možnosti aplikace na implementovaný informační systém.

Moderní řízení projektů v sobě skrývá nejedno úskalí a neobejde se bez aplikace promyšlené a v praxi ověřené metodiky a nástrojů. Osobnost vedoucího projektu, kromě splnění nutných charakterových vlastností a orientace ve věcné problematice projektu, musí zvládat velmi dobře činnosti projektového manažera. Tyto činnosti reprezentují praktické naplnění jednotlivých znalostních oblastí projektového řízení, jakými jsou např. řízení integrace projektu, řízení rozsahu, času, nákladů, rizik, kvality, nákupu atd. Realizace velkých projektů často klade vysoké nároky i na komunikační, řídící a kontrolní činnosti v rámci jednotlivých členů týmu.

2 Projektové řízení

Jako lidé řídíme vědomě či nevědomě různé projekty, kdykoliv děláme něco, co má předem daný cíl, začátek a konec. Někteří z nás jsou v tom přirozeně dobří, jiný si počínají hůře. Úspěšnost svých projektů může ale každý zvýšit přenesením pozornosti a důrazu na způsob, jakým jsou tyto projekty řízeny [18].

Lidé vždy byli, jsou a budou schopni úspěšně realizovat své projekty všeho druhu. Tuto schopnost ale bohužel nemá každý a dobré řízení projektů není jednoduché. Tento aspekt má za následek to, že se projektové řízení vyvinulo v samostatnou disciplínu, která v posledních dekádách systematicky zkoumá mnoho úspěšných i neúspěšných projektů. Výsledkem tohoto zkoumání jsou mnohá doporučení, ze kterých časem vzešly ucelené metodologie pro zdárné vedení projektů od začátku do konce. Řízení projektů jako samostatná disciplína v oblasti teorie řízení zaznamenalo v posledních dvou desetiletích výrazný rozvoj završený standardizací v mezinárodně uznávaných normách [11]. S globalizací společnosti a trhu v některých zemích bývá dokonce jejich užití podmínkou výběru dodavatele pro státní či firemní zakázky [18].

2.1 Definice projektového řízení

Projektové řízení představuje způsob rozplánování a realizaci složitějších, zpravidla jednorázových akcí, které je potřeba uskutečnit v požadovaném termínu s plánovanými náklady tak, aby se dosáhlo stanovených cílů. Jde o proces, ve kterém jednotlivci nebo organizace využívají své zdroje k realizaci projektů. Stručně můžeme řízení projektů také charakterizovat jako účinné a efektivní dosahování významných změn.

Projekt je činnost s jasně definovaným cílem, začátkem a koncem. Protože se vymyká běžné denní praxi, tak není většinou předem jistý jeho výsledek. Výsledek projektu může být hmotný i nehmotný. Například uspořádání realizace stavby, naplánování rozpočtu, redesign webové prezentace nebo rekonstrukce bytu jsou vše příklady projektů různých rozsahů.

Metodologie projektového řízení představuje způsob řízení projektu. Tato metodika může být buď přejatá a nebo vlastní, upravená na míru osobním potřebám. Metodologií ovšem nenazýváme intuitivní přístupy řízení, protože jsou ve své podstatě nahodilé a tudíž neopakovatelné, nedefinovatelné a prakticky nesdělitelné.

Primárním cílem každého projektu je jeho úspěšné dokončení. Projekt je však třeba dokončit v očekávané kvalitě, rozpočtu a čase. V praxi řízení projektů je většinou poměrně velká pozornost věnována metodické a procesní stránce projektového řízení a plánování projektu. Naopak často je však opomíjen systém řízené komunikace a správa projektové dokumentace, což má za následek obtížné začlenění nového pracovníka do týmu. Především v případech, kdy je důležitá komunikace ohledně specifikace zadání mezi projektovým manažerem a vývojářem vedena privátním způsobem, například elektronickou poštou. Má-li pak nový člen týmu navázat s prací tam, kde zkončil jeho předchůdce, nebude se mít čeho chytit. Bude si muset vyžádat tuto soukromou komunikaci, nebo se znovu doptávat vedoucího projektu, tak jako jeho předchůdce. V této situaci by zcela jistě pomohlo předejít těmto nutnostem zavedením nějakého centrálního místa, kde by

jednotliví členové týmu mohli zdokumentovávat průběh prací a proběhlou komunikaci ohledně projektu. Na základě tohoto poznatku může poté projektový manažer zdokonalit vedení projektu.

Kvalitní projektové řízení tedy ušetří mnoho času a sil při dosahování našich cílů. Čím častěji projekty řídíme a čím jsou tyto projekty větší a dražší, tím důležitější je jejich úsporné a efektivní řízení. Zkušený projektový manažer pak ovládá hned několik metodologií projektového řízení a na základě konkrétních specifikací a požadavků na projekt se dokáže rozhodnout a vybrat tu nejvhodnější.

2.2 Výhody lepšího řízení projektů

Při posuzování úspěšnosti projektu je zcela nejpodstatnější faktor, zdali byl vůbec zdárně dokončen. Dalšími kritérii jsou pak vynaložený čas, náklady a kvalita výsledku, kterou se zabývají disciplíny jako systémy managementu jakosti. Zde právě bývají u mnoha projektů značné rezervy. Lepším projektovým řízením tedy můžeme dosáhnout úspory peněz, mít více volného času a nebo zvýšit kvalitu výsledného produktu [18].

Znalost kvalitní metodologie může dodat projektovému manažerovi sebedůvěru pro zvládnutí větších projektů a zároveň zajistit značnou konkurenční výhodu u zadavatelů, kteří již mají s projektovým řízením zkušenosti. Další výhody se pak týkají hlavně organizací a pracovních týmů. Zavedení srozumitelné metodologie přináší opakovatelné metody a postupy, které se snáze sdělují novým členům projektového týmu a přispívají k lepšímu rozdělení pravomocí i zpřehlednění dokumentace.

S nárůstem intenzity používání projektového přístupu k řízení se ale ukazuje, že výrazně narůstá i důležitost podpory řízení projektů ze strany informačních technologií, zejména vhodně vybraným a kvalitním softwarem. Ten sice nemůže nahradit know-how faktory, může je ale podstatně umocnit. Je zřejmé, že již nevystačíme s tradičními nástroji podporujícími lokálně práci jednotlivých projektových manažerů [11].

Dnešní požadavky na moderní prostředí pro řízení projektů zahrnují pokrytí úplného životního cyklu projektů, včetně například sběru, vyhodnocení a schvalování nových námětů a požadavků, sestavení a vyhodnocení projektového (respektive investičního) portfolia, maximální automatizace administrativních a schvalovacích procesů pomocí workflow, notifikačních a eskalačních mechanismů, správy projektové dokumentace či podpory spolupráce projektového týmu [11].

Samozřejmostí je kvalitní podpora „tradičních“ kategorií, jako je samotné rozplánování úkolů projektu, správa projektových zdrojů a kapacitní plánování, finanční plánování projektu včetně zpětné vazby o skutečně realizovaných nákladech a výnosech, at'již formou faktur, nebo nepříliš populárních výkazů práce účastníků projektu, tzv. timesheets [11].

Častým požadavkem náročnějších softwarů je také možnost integrace s okolními systémy (ERP, intranet, service desk atd.) dané společnosti formou online nebo offline rozhraní. Požadavky na takové robustní systémy pro podporu řízení projektů nezůstaly stranou pozornosti významných dodavatelů softwaru, a proto již dnes většinou nabízejí ve svém produktovém portfoliu systémy, pokrývající více či méně kompletní životní cyklus projektů. Pro tuto kategorii softwaru je používáno označení „Enterprise PPM“,

které má zdůrazňovat právě schopnost podpory všech nebo alespoň podstatné většiny procesů multiprojektového řízení [11].

2.3 Zásady úspěšných projektů

Úspěšné projektové řízení je v případě nezávislých členů týmu závislé nejen na níže uvedených zásadách, ale zejména na dobrém hospodaření s časem. Následující body se mohou zdát jednoduché a jasné, ale hlavně u větších projektů může být obtížné všechny dodržet [18].

1. **Cíle projektu** – Čeho má projekt dosáhnout? Je projekt realizovatelný? Chceme jej realizovat? Neexistují nějaké lepší alternativy?
2. **Podpora zadavatele** – Máme jasnou podporu zadavatele projektu? Jsou ujasněny všechny závazky k dodání potřebných zdrojů (peněz, lidí, vlastního času aj.)?
3. **Spolupracovníci** – Je pro realizaci projektu k dispozici dostatečně spolehlivý a kvalitní tým?
4. **Pravomoce a odpovědnost** – Kdo bude koordinovat a kontrolovat práci ostatních? Je jasně rozdělena odpovědnost za dílčí činnosti na projektu i za projekt jako celek?
5. **Sledování kvality** – Jsou určeny mechanismy, jimiž bude při realizaci projektu sledována kvalita budoucího výsledku projektu, popř. i průběžných dílčích výstupů?
6. **Rozdělení na etapy a revize projektu** – Nebylo by vhodné projekt rozčlenit na několik fází, které by se snáze řídily a po jejichž skončení bychom vždy projekt přehodnotili oproti původnímu zadání?
7. **Rizika** – Zvážili jsme pečlivě všechna možná rizika projektu? Jsou tato rizika únosná? Víme jak tato rizika budeme řídit?
8. **Zainteresované strany** – Identifikovali jsme všechny zainteresované strany, tj. skupiny lidí a organizace, kterých se projekt dotýká? Víme jak mohou projekt ovlivnit a jak s nimi budeme komunikovat?
9. **Ostatní činnosti a projekty** – Když budeme realizovat tento projekt, nebudeme toho mít už celkově hodně? Nekoliduje s ostatními plány činností, jinými projekty?
10. **Vůdcovství** – Má projekt jasného vůdce a vizionáře?
11. **Dokumentace** – Jak bude vedena dokumentace projektu a jeho výsledného produktu k předání? Umožní nám dokumentace se v budoucnu k projektu jednoduše bez dalších studií vrátit?
12. **Závěrečná rekapitulace** – Co jsme se u projektu naučili, jakým chybám můžeme příště předejít?

2.3.1 Týmová spolupráce

Úspěšnost projektů zdaleka nezávisí jen na schopnostech samotného projektového manažera a využívaných podpůrných nástrojů. Jednou z podstatných charakteristik úspěšnosti projektů je schopnost týmové spolupráce jeho členů. Pracovní skupina neboli tým má význam nejen pro samotný projekt, ale i pro každého jedince. Schopnost týmové spolupráce se objevuje mezi nejčastěji uváděnými požadavky v pracovních inzerátech. Také týmový duch v různých podobách je velice často proklamovanou součástí firemní kultury. V oblastech, kde dominuje hlavně odbornost zaměstnanců, patří týmová spolupráce k největším uvědomovaným rozvojovým potřebám.

Umění týmově pracovat je jednou z mnoha dovedností, na kterou se v dnešní době klade velký důraz. Dobře sestavený a fungující tým dosáhne mnohem více než samotný jedinec nebo špatně fungující skupina. Tým, který pracuje podle principů týmové spolupráce, dokáže úspěšněji a efektivněji dosahovat stanovených cílů. Umění spolupracovat je již pomalu považováno za běžnou dovednost, která je čím dál více vyžadována, ale pokud má být tým úspěšný a dosahovat vynikajících výsledků, musí existovat v týmu taková efektivní týmová spolupráce, na níž jedinci budou chtít participovat. Změny, které jsou součástí našeho profesního i osobního života, vyžadují zapojovat se do nových vztahů a spolupracovat s dalšími lidmi.

Týmová spolupráce využívá tvořivosti lidí, jejich rozdílných schopností a umožňuje dosahovat výkonů, kterých by jako jednotlivci často nebyli schopni dosáhnout. Vytvoření funkčního týmu je dlouhodobý proces založený na vzájemném respektování, poznání, důvěře, vyvážených vztazích a jednoty v cílech, kterých chce dosahovat. Pro kvalitní práci týmu je nezbytná otevřená komunikace všemi směry. Dále je vhodné, aby se každý člen ujal dle svých kompetencí[1] specifických funkcí (rolí), nutných pro efektivní fungování týmu a přispíval ke splnění společného úkolu týmu svými odbornými znalostmi. Jen tak má tým příležitost, aby při řešení úkolů, problémů a rozhodování efektivně využíval svoje zdroje a dosahoval tak vysoké produktivity.

2.3.2 Charakteristiky týmové spolupráce

Klasická práce ve skupině se od týmové práce liší hned v několika bodech. Počet členů skupiny je neomezený, důležitá je především jejich koordinace [12]. Dosahování cílů jednotlivých členů větší skupiny většinou nezávisí na ostatních členech skupiny. Díky tomu mohou členové skupiny vykonávat práci samostatně. Narozdíl od menších skupin (obvykle 5 až 9 členů) jsou členové týmu z hlediska dosahování dílčích cílů a výsledného cíle na sobě závislí. Mezi další významné rozdíly patří specifické metody pro řešení projektů a rozdělení kompetencí jednotlivých členů týmu [12].

2.3.3 Týmové role a interprofesionální vztahy

Každý z členů týmu vystupuje v různých oblastech, které charakterizují jeho vlastnosti. Stejně tak jako je každý z nás v běžném životě úspěšný, či neúspěšný v různých oblastech

[1] Jednotlivé kompetence pro týmovou spolupráci (role) rozebrány v podkapitole 2.3.3

působení, tak jsou i členové týmu vhodní, či méně vhodní do různých oblastí týmové spolupráce. Tyto oblasti se označují jako role a charakterizují přirozené vlastnosti a cíle každého jedince.

V roce 1996 Dr. Meredith Belbin ve své knize [2] představil výsledky práce na Henley Management College zkoumající úspěšnosti týmů. Identifikoval devět různých typů chování, každý z nich se nazývá týmovou rolí. Každá týmová role má svou vlastní kombinaci přínosů a přípustných slabin.

2.3.3.1 Inovátor Inovátor je kreativní, neortodoxní individualista, který přichází s novými nápady a originálními řešeními. Disponuje velkou představivostí a je zdrojem inspirace pro celý tým. Dokáže celý projekt vymyslet a rozjet, ale není dobrý vůdce. Je u něj také velká pravděpodobnost, že jej nedotáhne do zdárného konce.

2.3.3.2 Vyhledávač Je extrovertní osobnost, dobrý improvizátor, povahou společenský a přátelský, s vřelými kontakty. Ví na koho se může obrátit při shánění všech podstatných informací, na kterých je tým závislý, proto je důležitý především v počátečních fázích projektu. Vidí v nápadech inovátora příležitosti a ví jak je využít. Dokáže tým motivovat a prezentovat jeho výsledky, ale je už méně schopný samotné práce. Má proto prst pevně na tepu týmu z vnějšího světa, ze kterého mu také zajišťuje dostatečnou podporu.

2.3.3.3 Koordinátor Koordinátor nebo-li režisér je role dozorčí funkce v týmu, bývá dobrý vůdce, který posouvá rozhodnutí směrem dál. Nastavuje hranice a pečlivě střeží jejich dodržování. Dohlíží na to, aby měl každý prostor pro vyslovení svého názoru, vysvětluje rozhodnutí. Především však dohlíží na to, aby vše probíhalo podle plánu. Uvědomuje si schopnosti jiných a má cit pro delegování úkolů na správného člověka. Dokáže také ostatním pomoci soustředit se na jejich práci, ale občas je pro svou snahu delegovat všechny úkoly na jiné členy týmu vnímán jako manipulativní a vyhýbající se práci.

2.3.3.4 Formovač Úzce zaměřený vůdce, který oplývá soutěživostí a vysokou motivací pro dosažení cílů, a velmi často se mu je daří splnit. Je vnitřně zavázán k dokončení cílů, nerad nechává něco nedokončeno či neuzavřeno. Dává tvar a obsah práci a podněcuje své svěřence k lepším výkonům. Dle Belbina [2] může přítomnost dvou čí více formovačů v týmu vést ke konfliktům, zhoršení vztahů či až k soutěžení formovačů navzájem mezi sebou.

2.3.3.5 Vyhodnocovač Je spravedlivý a realistický pozorovatel či kritik, posuzující činnost týmu. Povahou je racionální a trochu skeptický. Je nejobjektivnějším členem týmu, hodnotí nápady, poskytuje jejich detailní analýzy a varuje před možnými chybami. Díky svým vlastnostem dokáže předcházet chybám, proto je jeho přítomnost v týmu důležitá. Nedokáže ale inspirovat ostatní členy týmu a často je svou kritikou demotivuje. Na některé členy týmu také může působit provokativně a jako člověk, který ničemu nevěří.

2.3.3.6 Týmový pracovník Tato role bývá občas v literatuře označována jako podporovatel. Je jednou z největších opor týmu. Stmeluje tým dohromady a zajišťuje jeho chod. Je vnímavým posluchačem, který je schopen jednotlivé nápady rozvinout do vyšších sfér, všímá si potřeb a problémů ostatních. Svou povahou a chováním pomáhá vytvářet dobrou atmosféru v týmu, snižuje napětí v konfliktních situacích, např. pomocí humoru. Plně se ztotožňuje s týmovými cíly a pro samotný tým a jeho správnou funkci je schopen obětovat téměř vše. Je ale často nerozhodný a neměl by tak dělat klíčová rozhodnutí, na kterých závisí další činnost týmu.

2.3.3.7 Realizátor Disciplinovaný, spolehlivý, konzervativní a neúnavný tahoun celého týmu. Stará se o to, aby jednotlivé nápady a rozhodnutí bylo možné převést na konkrétní případy, orientuje se tedy na možnost jejich realizace. Pracuje systematicky, vytváří pevné struktury, je svědomitý a disciplinovaný. Je motivován loajalitou k týmu či firmě, ve které pracuje. Špatně se proto přizpůsobuje novému prostředí.

2.3.3.8 Dotahovač Stará se o to, aby se dodržoval časový plán. Perfekcionista, který do detailů vše kontroluje. Dává pozor, aby se na nic nezapomnělo. Svou práci pečlivě a svědomitě dokončí za každou cenu. Zřídka potřebuje povzbuzení od ostatních členů týmu, protože ho motivují především jeho vlastní vysoké standardy, kterých se snaží dosáhnout či překonat. Mívá však občas přehnané obavy, svou práci nesvěří někomu jinému a váhá s přidělováním úkolů.

2.3.3.9 Specialista Je cílevědomý, oddaný, iniciativní odborník, který je úzce profilován a zcela zaměřen pouze na svůj obor. Má hlubokou, ale úzkou znalost věci a neustále zlepšuje své vědomosti. Existuje-li otázka, na kterou nedokáží odpovědět, ihned se pustí do hledání odpovědi. Specialisté přinášejí týmu vysokou úroveň koncentrace, schopnosti a dovednosti ve svém oboru. Přispívá ale pouze k záležitostem týkajících se jeho oboru a k ostatním věcem ležícím mimo jeho úzké hranice je nezaujatý.

Hodnota teorie týmových rolí Belbina spočívá v tom, že umožňuje jednotlivci i týmu přizpůsobit se vnějším požadavkům a využít sebepoznání k úspěchu. Týmová role v tomto pojetí je zčásti projevem osobnosti, vrozených povahových vlastností a získaných životních zkušeností, ale z části také projevem aktuální situace na pracovišti.

Optimálně složený tým obsahuje všechny typy rolí. To neznamená, že by v týmu muselo být vždy devět lidí. Belbin připisuje vyšší efektivnost týmu o třech, čtyřech nebo pěti lidech za předpokladu, že každý ze členů bude vykonávat současně několik rolí. Pokud nějaká role chybí, projeví se to negativně na výsledcích týmu.

Lidé jsou spíše kombinací několika typů rolí, což odpovídá velké různorodosti lidských povah a přístupů k práci. Vedle hlavní role má pak každý jedinec v týmu ještě jednu nebo dvě role záložní. Pokud postrádáme týmového pracovníka, měl by se této role automaticky chopit jedinec, který k ní má nejblíže. Některé role si ale ideologicky i prakticky odporují a jsou neslučitelné proto bychom pracovníka k nim neměli nutit i kdyby měl k roli nejblíže [5].

2.3.4 Výhody a nevýhody týmové práce

K hlavním výhodám týmové spolupráce patří výrazné zvýšení produktivity práce v oblastech, které vyžadují kreativní řešení různých problémů a určitou míru přizpůsobivosti. Budování shody mezi členy týmu vylučuje extrémní postoje a názory. Jednotliví členové týmu se mohou ke správnému řešení vzájemně inspirovat. Spolupráce mezi členy týmu zlepšuje výrazným způsobem komunikaci. Může tak posílit důvěru a vzájemnou podporu při plnění pracovního úkolu [13].

Tým je schopen v určitých situacích přijmout lepší rozhodnutí než jednotlivec. Zdá se také, že společně sdílená odpovědnost za riskantnější rozhodnutí, má za následek snížení pocitu osobní zodpovědnosti v případě, že by se rozhodnutí ukázalo jako nesprávné. Nevýhody týmové práce jsou víceméně subjektivního charakteru. Pro nové členy nemusí být z počátku jednoduché zvyknout si na nové postupy, jednat podle stanovených pravidel a vzorců, neboť se mohou domnívat, že jejich myšlenka, či nápad je nejlepším řešením a význam týmové spolupráce jim tak uniká.

3 Osobní produktivita a organizace práce

Být z osobního hlediska produktivní dnes pro mnoho lidí znamená zanechávat za sebou dobré výsledky, rozvíjet své schopnosti či splnit více povinností za méně času. Vedlejším efektem této činnosti často bývá velké stresové vypětí, ztráta motivace nebo nechuť k práci. Mezi faktory ovlivňující produktivitu práce patří kvalifikace a motivace pracovníka, použité technologie a v neposlední řadě organizace řízení.

Mnoho lidí pracuje pouze pro peníze. A to je důvod, proč je jejich pracovní produktivita (nejen pracovní) velice nízká. Vidět ve své práci smysluplný účel motivuje k nalezení energie a síly pro pokračování v práci, i přestože se práce nedaří. Právě proto, že v práci vidíme něco „hlubšího“ než peníze.

Dalším důležitým prvkem pro vysokou osobní produktivitu je definice cílů. Jestliže mám přesně stanovený cíl, dokážu si zvolit vhodnou strategii pro splnění právě tohoto cíle. Člověk, který nezná své cíle velice neefektivně využívá svůj čas.

Nutné je si také dobře zvolit priority. Věnovat se v první řadě věcem, které jsou nejdůležitější, až poté, co splníme tyto věci, můžeme plnit další povinnosti.

Také je velmi vhodné znát osobní denní produktivitu práce. Každý člověk je jedinečný a má různé potřeby. Ale je známo, že po probuzení výkonnost člověka stoupá a těsně před polednem dosahuje svého maxima. Samozřejmě záleží na tom, v kolik hodin člověk vstává a jak dlouho mu trvá, než se „nastartuje“ jeho organizmus. Poté produktivita lehce klesá. Ve večerních hodinách efektivita práce s přibývajícím časem strmě klesá. Dále je doporučeno začlenit přibližně ve dvou hodinových intervalech menší přestávky či střídání fyzické činnosti s mentální činností.

Shrnu-li předchozí odstavce, být více produktivní je vlastnost, která se dá za pomocí osvědčených postupů a rozvíjením osobnosti výrazně zlepšovat. Jak se stát více produktivním? Je třeba se soustředit pouze na jednu konkrétní věc, eliminovat faktory vyrušování, naučit se lépe organizovat svou práci, nápady nenechávat v hlavě, ale zaznamenávat je. K tomuto všemu se dají využít moderní technologie.

Existuje mnoho časem a také praxí ověřených postupů vedoucích ke zvýšení osobní produktivity. Většina z nich zahrnuje osvojení dovednosti plánování času neboli time managementovu. Mezi výkonnými manažery a často i mezi IT profesionály se v posledních letech stala velmi oblíbenou metoda Getting Things Done, která je částečně tématem této bakalářské práce.

3.1 Metoda Getting Things Done

Getting Things Done je metoda organizace práce vytvořená americkým koučem Davidem Allenem, popsána ve stejnojmenné knize. Název metody je často zkracován jako GTD. Narozdíl od jiných time managementových metod se orientuje především na kroky spojené s řízením pracovního procesu.

Základní myšlenkou k dosažení vyšší osobní produktivity je zaměřit své soustředění na jednu jedinou věc, která je v daném okamžiku důležitá. Metoda dále obsahuje ucelenou sadu postupů, jak věci dělat, kdy je dělat a jak určit pořadí, ve kterém úkoly zpracovávat. A protože lidský mozek není podle Allena uzpůsoben k připomínání úkolů, schůzek

a všech závazků, doporučuje vytvořit si externí systém různých druhů seznamů, které následně poslouží jako připomenutí, a dostat tak veškeré tyto rušivé elementy z hlavy. Jedině takto je možné se opravdu produktivně soustředit na jeden konkrétní úkol [1].

David Allen ve své knize zmiňuje, že neexistuje žádné univerzální řešení, které by fungovalo pro každého a za všech okolností. Dodává ale, že existují postupy a doporučené návyky, které nám to mohou usnadnit [1].

3.1.1 Požadavky pro zvládání úkolů

Dle metodiky Getting Things Done je třeba si k zvládání úkolů osvojit některé ze základních návyků. Jedním z nich je udržovat čistou mysl. Vše, co pokládáme za nedokončené či nějakým způsobem nevyřešené, je třeba zachytit v důvěryhodném systému mimo naši hlavu. K tomuto shromaždišti se musíme pravidelně vracet a třídit jeho obsah. Dále si musíme přesně ujasnit obsah svého úkolu, jednotlivé dílčí kroky, jeho cíle a rozhodnout se, co je třeba udělat, abychom pokročili k jeho naplnění. Všechny tyto dílčí kroky je pak nutné uspořádat si do našeho důvěryhodného systému.

Dokud nám větší množství nedokončených úkolů leží v hlavě, náš mozek se jim stále věnuje. Stále na ně podvědomě myslíme a takto vzniká pocit, kdy si myslíme, že nestíháme své závazky plnit. Ošálit můžete kohokoliv, svou mysl ale neoklamete. Ta pozná, zda jste došli k potřebným závěrům a zda jste si výsledky a jednotlivé kroky uložili na místo, kde je spolehlivě najdete. Jestliže jste tak neučinili, vaše hlava nepřestane pracovat přesčas. Dokonce i když jste se už rozhodli, jaký další krok k vyřešení problému podniknete, vaše mysl nepovolí, dokud si nevytvoříte připomínku na místě, o kterém víte, že se tam nepochybně podíváte. Bude na vás ohledně tohoto neprovedeného kroku neustále tlačit, a to obvykle v době, kdy s ním nic dělat nemůžete, což váš stres dále znásobí [1, str. 23].

3.1.2 Model přirozeného plánování

Allen ve své knize popsal několik modelů plánování (přirozené, nepřirozené a reaktivní), avšak právě model přirozeného plánování dál rozvádí a je na něm založen základ pro zbytek metodiky Getting Things Done.

Tento model vychází z jednoduchého faktu, že nejzkušenějším plánovačem na světě je lidský mozek. Aniž si to uvědomujeme, plánování nás provází každým dnem. Může jít o zcela samozřejmé věci jako například nákup v obchodě či telefonní hovor. Jelikož jde o proces nahodilý a intuitivní, dle definice projektového řízení v podkapitole 2.1, nemůže být tedy považován za metodologii. Nicméně Allen si uvědomil, že i u těchto zcela samozřejmých činností dělí myšlenku o vykonání od samotného vykonání posloupnost kroků, kterými mysl prochází, jen si je většinou neuvědomujeme. Formuloval tedy pět kroků, kterými mysl prochází při projektovém plánování. Takovýto nadefinovaný a opakovatelný přístup řízení již může být považován, dle definice projektového řízení 2.1, metodologií [1].

3.1.3 Pět kroků projektového plánování

3.1.3.1 Definice cíle a zásad Přemýšlení nad výstupy je jedním z nejefektivnějších prostředků, jak proměnit přání v realitu. Je třeba si nejdříve ujasnit záměr projektu, důvod realizace a také zásady, které je potřeba dodržet. Jsou-li splněny tyto předpoklady, pak má následné plánování projektu jasný rozměr i formu, což nám usnadní další práci. Pokud se při plánování či samotném řízení projektu stane, že nevíme jak dále, je třeba se znovu vrátit ke stanoveným cílům a záměrům.

3.1.3.2 Ujasnění výsledků Každá cesta k reálným výsledkům vede nejprve přes mysl člověka. Vytváření jasných výsledků vede k tomu, čeho chceme dosáhnout a ukazuje nám i cestu jak toho dosáhnout. Proces soustředění na scénář úspěchu projektu obvykle doprovází pocit intenzivního nadšení a přináší jedinečné pozitivní nápady, na které doposud nebylo pomysleno [1].

3.1.3.3 Brainstorming Jakmile máme ujasněnou představu, čeho chceme dosáhnout, přichází na řadu zjistit, jak toho dosáhnout. Účelem brainstormingu je zaznamenat všechny příchozí myšlenky, které se pojí s budoucím projektem, i když nejspíše budou velmi nahodilé a nesouvislé. Zde není důležitá kvalita myšlenek a nápadů, ale spíše jejich kvantita, která dovolí rozlet našemu myšlení. Nepřiměřené kritizování či zpochybňování myšlenek vede k útlumu tvůrčího procesu. Častým a velmi populárním nástrojem k vizualizaci myšlenek jsou graficky orientované myšlenkové mapy, někdy též nazývány jako mentální mapy.

3.1.3.4 Uspořádání Následujícím a bezprostředním krokem po brainstormingu je uspořádávání všech nápadů. Jakmile dostaneme nápady z hlavy a máme je před očima, všimneme si jejich vztahů, struktury a posloupností. Výstupem této fáze je jasná představa, co vše je třeba učinit pro úspěšný výsledek projektu, zjistit které prvky jsou nejdůležitější a určit jejich priority a posloupnosti v čase. Nejdůležitější prvky často určují milníky projektu. Jakmile je stanovena základní kostra projektu, je na řadě vyplnit mezery a dát tak projektu úplný tvar, což je náplní poslední fáze projektového plánování.

3.1.3.5 Určení dalších kroků Posledním krokem plánování je určení prvních konkrétních činností, které lze nyní udělat, aby se projekt posunul dopředu. Pokud nemáme konkrétní odpověď, je potřeba podrobněji promyslet některou z předchozích fází posloupnosti přirozeného plánování.

3.1.4 Pět fází řízení pracovního procesu

3.1.4.1 Sbírání Základním krokem pro zvládání úkolů, je shromáždění všech záležitostí, které vyžadují naši pozornost, do předem stanovených sběrných míst, tzv. schránek[2].

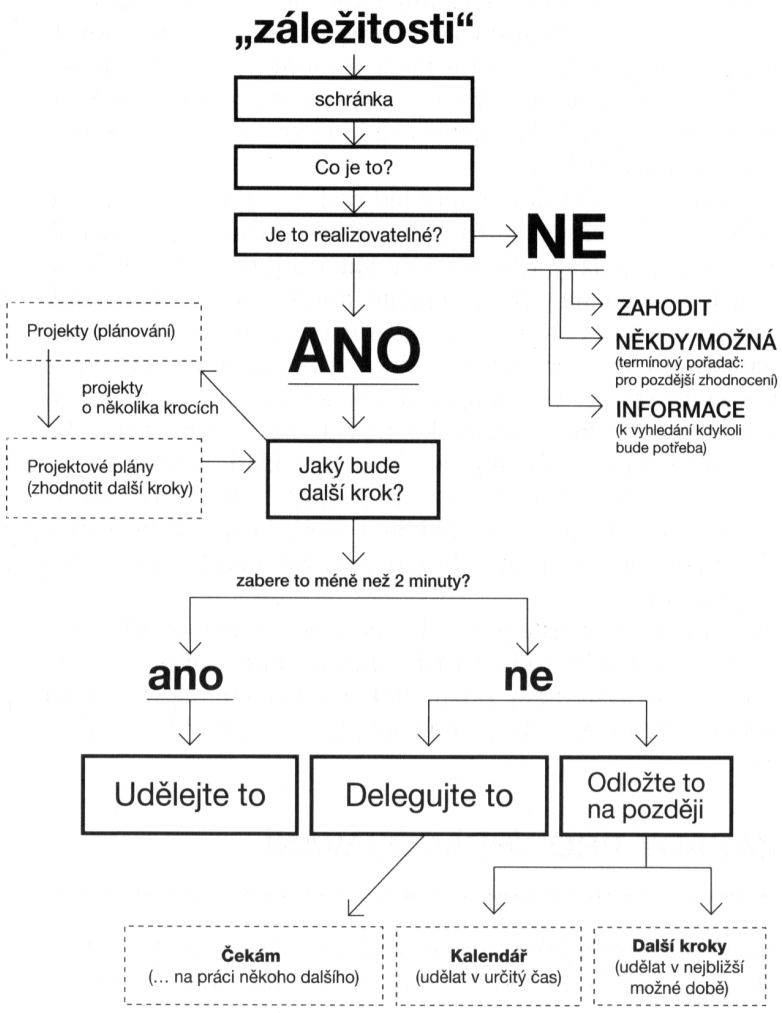
[2]Za schránku lze považovat prakticky cokoliv co slouží k shromažďování dat, např. složka nepřečtené pošty e-mailového klienta, přihrádka na papíry nebo došlou poštu, PDA či hlasový záznamník

Mezi tyto záležitosti patří jak hmotné tak i nehmotné záležitosti. Mezi hmotné záležitosti například patří poznámky a připomínky na pracovním stole, studijní materiály, které je potřeba zpracovat, obsah přihrádek v pracovním stole, kde se mohou nacházet důležité či zapomenuté dokumenty jako rozpočty projektů či manuály k prostudování. Nehmotné záležitosti se nacházejí v mysli. Je třeba si vybavit a zapsat každou důležitou myšlenku, každý nápad, všechny závazky, sliby, pracovní záležitosti či e-maily, na které je třeba reagovat. Shromáždění těchto záležitostí vede k upřesnění představ o objemu záležitostí, které je třeba zvládnout, a vede k většímu pocitu kontroly nad svými úkoly. V této fázi nemá smysl se zabývat tím, co jednotlivé záležitosti znamenají. Cílem sběrného procesu je pouze dostat co nejrychleji do vstupů vše, co tam patří. Dalším krokem je zpracování těchto vstupů, což vede k pravidelnému vyprazdňování vstupní schránky.

3.1.4.2 Zpracování Další fáze procesu zahrnuje zpracování obsahu schránek. Zpracování obsahu schránek v této fázi neznamená splnění všech úkolů, projektů a kroků, které s nimi souvisí, ale rozhodnutí, co je potřeba s každou položkou udělat. U každé položky ve schránce je potřeba zvážit, zda je v daném okamžiku realizovatelná, nebo není. Nerealizovatelné záležitosti se přímo odstraní, nebo se uloží pro případnou realizaci mimo schránku pro pozdější zhodnocení. Žádná položka se však do schránky nevrací zpět. Realizovatelné záležitosti se buď vykonají, delegují na někoho jiného, nebo odloží pro pozdější zpracování. O odložení či vykonání rozhoduje tzv. pravidlo 2 minut. Jde-li záležitost provést během dvou minut, nikam se neodkládá a ihned se vykoná během zpracování schránky. Úkoly, které nelze řešit jedním krokem, je doporučeno rozdělit na více kroků a vytvořit z nich projekt. Cílem fáze zpracování je rozhodnout se, co věc znamená a jaký krok je nutný, a podle toho ji vyřídit. Celý postup procesu zpracování je znázorněn na obrázku 1.

3.1.4.3 Organizace V třetí fázi procesu Allen učí zařadit již zpracované položky na pět různých seznamů připomenutí, které jsou klíčem celého dalšího fyzického řízení.

- **seznam projektů**, což jsou záležitosti s očekávaným výsledkem, které se musí realizovat více než jedním krokem a u kterých je vždy definován další krok
- **seznam dalších kroků** zahrnuje položky, které je třeba vykonat co nejdříve v nejbližších volých chvílích
- **seznam věcí, na které se čeká**, nebo-li delegované akce, u nichž se čeká, až je provedou jiní
- **kalendář** obsahující činnosti, které se musí provést v určitém čase nebo dni
- **nerealizovatelné položky**, o nichž člověk v okamžiku zpracování nedokázal rozhodnout, zda je v budoucnu bude chtít realizovat, či položky nevyžadující žádnou akci spadající do kategorií *položky k dozrání*, *informace určené k archivaci* či *k uložení pro pozdější přehodnocení*



Obrázek 1: Schéma pracovního procesu dle metodiky GTD [1, str. 39]

Projekty jsou vedeny na samostatném seznamu projektů, který je nutné pravidelně procházet a přehodnocovat. Není nutné je organizovat podle rozsahu nebo priority, je ale požadováno mít u každého projektu definován další krok [1].

Jednotlivé položky na seznamu dalších kroků nejsou nijak vázány ke konkrétnímu dni či času vykonání. Seznam obsahuje soupis předem definovaných akcí, do nichž je možno se pustit ve volných chvílích během dne. Je doporučeno jednotlivé akce opatřit kontexty, s jejichž pomocí je můžeme uspořádávat a na jejich základě můžeme selektivně vybírat akce týkajících se stejných či podobných kontextů. Jedna věc je zapsat si, že potřebuji koupit komponentu počítače a druhá vzpomenout si na to ve městě. Pokud jsem si schopen vyhledat všechny akce, které spolu sice nemusí nijak souviset, ale které mohu splnit, nacházím-li se zrovna ve městě, mohu tak ušetřit čas a další případnou cestu do města.

3.1.4.4 Zhodnocení V této fázi je potřeba vytvořit si návyk jednou týdně projít všechny nedokončené úkoly, otevřené záležitosti, kalendář a projekty a rozhodnout se, čemu je třeba se v dalších dnech věnovat. Vědět v každém okamžiku, co a kdy se musí udělat, uvolňuje prostor k flexibilitě. Týdenní zhodnocení poskytuje možnost k sesbírání a zpracování všech věcí k vyřízení, revizi systému, aktualizaci a doplnění seznamů. Většina lidí se v práci cítí nejlépe, když uzavřou úkoly a upřesní závazky, které uzavřeli sami se sebou i s ostatními. Přesně k tomuto by měla fáze zhodnocení sloužit. Dobrý pocit poté motivuje člověka k další práci [1].

3.1.4.5 Vykonávání Úkolem poslední fáze řízení pracovního procesu je usnadnit rozhodování o tom, čemu se v každém časovém okamžiku budeme věnovat. Každé rozhodnutí jednat je intuitivní. Účelem této fáze je také změnit doufání, že jsme se rozhodli pro plnění správného úkolu, k přesvědčení, že toto rozhodnutí bylo správné.

V této fázi bychom měli mít nastavený systém, podle nějž rozhodujeme o tom, co se bude dělat. Jakýkoliv organizační systém je k bezcenný, pokud je k údržbě třeba vynaložit více času a úsilí než k samotnému plnění úkolů. Je proto třeba mít zavedený systém co nejjednodušší a měl by být přínosem a ne přítěží. Při rozhodování o tom, co budeme dělat bychom měli uplatňovat následující čtyři kritéria:

1. kontext, v němž se nacházíme
2. dostupný čas, který máme k dispozici
3. dostupná energie, která nám zbývá
4. priorita daného úkolu (*můžu, chtěl bych, měl bych, musím*)

3.1.5 Využití jako metodologie projektového řízení

Práce v informačních technologiích má narozdíl od jiných profesí nevýhodu, že není viditelná na první pohled. Není tak zřejmé, kdy je hotová a kdy není.

Problémem mnohých projektů je, že postrádají své hranice. Pracovníci pak v průběhu delší práce na těchto projektech ztrácejí představy o jejich cílech. Proto je důležité jasně vytyčit hranice a cíle, jinak podle Allena riskujeme, že práce s nejasnými hranicemi bude mít i nejasný tvar [1].

Výhodou metody Getting Things Done je její jednoduchost a dobrá použitelnost v praxi. Snadno se dá také rozšířit pokročilejšími technikami plánování. Je třeba ale mít na paměti, že slovo projekt má zde jiný význam než v klasickém projektovém řízení. Allen má značně volné pojetí slova projekt, kterým označuje jakoukoliv posloupnost několika na sebe navazujících kroků. Definice projektu z pohledu projektového řízení je přísnější, na užitečnosti této metody i pro skutečné projekty to nicméně nic nemění.

4 Architektura moderní webové aplikace

Webová aplikace je software založený na modelu klient-server. Klient v tomto případě představuje webový prohlížeč nebo HTTP agent v případě automatizovaných botů. Nejčastějšími zástupci ze strany webových prohlížečů jsou populární a rozšířené prohlížeče Mozilla Firefox, Internet Explorer, Google Chrome, Opera, Safari či Konqueror. Roli serveru zastane webový server s dostatečným aplikačním vybavením, což v případě webových aplikací nejčastěji znamená se službou aplikačního HTTP serveru, prostředím pro běh potřebného programovacího jazyka, ve které je webová aplikace napsána, a velmi často také přítomností některého ze SŘBD.

Výhodou webových aplikací oproti jejich destopovým protějškům je především centrální správa a schopnost aktualizace bez nutnosti šířit a instalovat software na potenciálně velkým počet klientů. Další nespornou výhodou je globální dostupnost a fakt, že jakékoliv zařízení obsahující webový prohlížeč a tedy i klienta pro naši aplikaci.

Webové aplikace samozřejmě mají i své nevýhody. Mezi ty hlavní patří výkonnost, objemný síťový provoz a doba odezvy u náročnějších aplikací. Dále je zde nepopíratelný fakt, že webové aplikace nemají přístup k API operačního systému jako aplikace desktopové. Především zde je největší propast mezi destopovými a webovými aplikacemi. Vše je otázkou klientů a tato bariéra bude dle mého názoru, podle současného vývoje a trendů, časem odbourána, především díky novému a velmi očekávanému standardu HTML 5 či moderním technologiím jako jsou Adobe Flex, Microsoft Silverlight, které dávají vývojářům platformu určenou pro tvorbu dynamického online obsahu a interaktivní práce s ním.

4.1 Web 2.0

Web 2.0 je označení pro etapu vývoje webu týkající se období od roku 2004 do současnosti. Poprvé byl tento výraz použit jako název konference pořádané v roce 2004 [19]. *Web 2.0* odpovídá v softwarovém průmyslu nové verzi, zásadně lepší oproti původní, a tak mělo toto označení být vhodnou metaforou pro „druhý dech“ internetového podnikání.

Klíčovými charakteristikami aplikací označovanými jako *Web 2.0* je orientace na uživatele. Uživatel by měl být vtažen do tvorby obsahu, a ta by měla být díky síťovému efektu s přibývajícím počtem uživatelů stále lepší a nabídnout jim tak kvalitní obsah.

Dalšími častými charakteristikami *Web 2.0* aplikací jsou: [19]

- sdílení a znovuvyužití informací
- otevřená komunikace
- organizovaný a roztříděný obsah s propracovanější hyperlinkovou strukturou
- reputační systémy
- rozmlžení hranice autor/čtenář
- možnost využívat API a vytvářet nové služby

Není nutné, aby aplikace splňovala všechny body této charakteristiky, aby mohla být označena jako *Web 2.0* aplikace. Málokterá totiž bude plně odpovídat všem charakteristikám, jde především spíše o změnu přístupu. *Web 2.0* znamená především maximální možné zaměření na uživatele [19]. Aplikace by také měla nabídnout uživateli maximální uživatelskou přívětivost.

4.2 Funkční požadavky moderních webových aplikací

Většina charakteristik *Web 2.0* popsaných v podkapitole 4.1 nepopisuje funkční požadavky samotné aplikace. Dle Tichého [17, str. 3] by měly pokročilejší webové aplikace, jako jsou *Web 2.0* aplikace, splňovat následující funkční požadavky, které jsou pro většinu pokročilejších webových aplikací společné a typické:

- vhodná architektura aplikace
- modularita
- zabezpečení aplikace
- validace vstupních parametrů
- uživatelské autentizace a správa uživatelů
- sessions a autentizace
- autorizace a přístupová práva
- jednotná databázová vrstva
- logování událostí a akcí
- validita výstupních stánek
- maskování URL

Tyto funkční požadavky by měl zajišťovat komplexní aplikační framework [17].

Hned prvním bodem funkčních požadavků moderních webových aplikací je „vhodná architektura aplikace“. Současně nejpoužívanějšími architektonický vzory jsou Model-View-Controller a v menším zastoupení v některých implementačních jazycích také Model-View-Presenter. Jelikož tato kapitola pojednává o architektuře jakékoliv obecné moderní webové aplikace, představím v následujících podkapitolách oba architektonické vzory. Pro implementaci analyzovaného informačního systému poté vyberu v podkapitole 6.1 vhodné implementační prostředí a aplikační framework, který bude nejlépe splňovat zmíněné požadavky kladené na aplikaci, a který bude podporovat právě MVC či MVP.

4.3 Model-View-Controller

Architektonický vzor Model-View-Controller (zkráceně MVC) pochází z konce 70. let minulého století. V roce 1979 jej představil Trygve Reenskaug jako architekturu pro vývoj interaktivních aplikací. Tento vzor se díky obrovskému rozmachu celosvětové sítě Internet v posledním desetiletí těší velké popularitě, kdy se do širšího povědomí vývojářů dostal zejména s rozšířením frameworku Ruby on Rails. Své kořeny má ale ve Smalltalku, kde se původně používal pro zakomponování tradičního postupu vstup–zpracování–výstup do GUI programů [17].

Vzor MVC rozděluje aplikaci do tří samostatných logických částí tak, aby je šlo upravovat samostatně a dopad změn byl na ostatní části co nejmenší. Tyto tři části jsou Model, View a Controller. Pro každou z nich vzor přesně definuje, za co je v rámci aplikace odpovědná.

4.3.1 Model

Model je v aplikaci zodpovědný za udržování stavu aplikace a zastřešuje aplikační (občas nazývanou také jako doménová) logiku. Model si lze tedy představit jako datový prostor a množinu různých stavů (i nekorektních), do kterých se aplikace může dostat. Požadované chování modelu vůči datům je, aby se model choval jako brána a zároveň i jako úložiště k datům. Měl by být zapouzdřený jako celek a pro View a Controller nabízet přesně definované rozhraní [17].

4.3.2 View

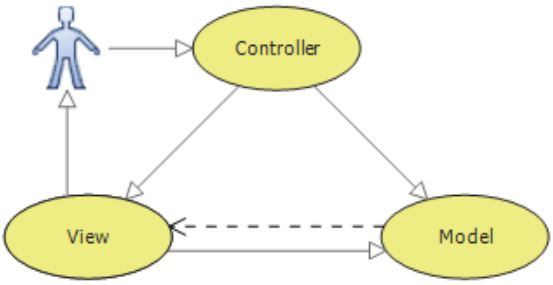
View jsou zodpovědné za generování uživatelského rozhraní, obvykle založených na datech z modelu. Pohled je tedy zobrazením modelu a dalších prvků uživatelského rozhraní [3]. U webových aplikací generuje View příslušný HTML kód, který je odeslán prohlížeči jako odpověď na požadavek. Jestliže slouží zobrazený výstup zároveň jako oblast pro zachytávání událostí od uživatele (například kliknutí myší do vykresleného okna), předává View tyto události Controlleru [17].

4.3.3 Controller

Controller má na starosti tok událostí v aplikaci a obecně aplikační logiku. Zpracovává veškeré vstupy a události pocházející od uživatele. Pro webové aplikace jsou hlavním vstupem HTTP GET nebo POST požadavky odeslané uživatelovým prohlížečem. Na jejich základě volá příslušné procesy Modelu, mění jeho stav apod. Podle událostí přijatých od uživatele i podle výsledků akcí v Modelu pak vybírá Controller vhodné View pro další zobrazení. Controller je tedy jakousi ústřední výkonnou jednotkou, která se stará o celkové provázání funkčnosti aplikace [17, 3].

4.3.4 Vazby a datové toky

Přímé vazby mezi jednotlivými částmi vzoru MVC existují jen dvě: [3]



Obrázek 2: Vazby mezi jednotlivými komponentami architektury MVC

- Controller má přímý odkaz na Model, aby mohl upravit jeho data
- View má přímý odkaz na Model, aby mohl jeho data zobrazit

Přímou vazbou mezi komponentami je myšleno, že si komponenta drží přímý odkaz na jinou. V moderních aplikacích ale často implementace vzoru v závislosti na konkrétní variaci MVC obsahuje přímou vazbu mezi Controllerem a View. Naopak nikdy nesmí existovat přímá vazba Modelu na ostatní dvě komponenty. Model nemůže držet přímý odkaz na View nebo na Controller [3]. Jednalo by se o hrubou chybu návrhu aplikace a tento fakt dodržují i konkrétní variace MVC.

Za nepřímé vazby jsou považovány například notifikační mechanismy, například pomocí vzoru Observer [4]. Nepřímé vazby se dají využít například právě u Modelu k upozornění konkrétních View o tom, že data Modelu byla změněna.

4.3.5 Výhody MVC

Mezi hlavní výhody využití tohoto vzoru patří: [7, 17]

- Snadné zpřístupnění pro různé druhy klientů. Pro zavedení podpory dalšího klienta je nutné nadefinovat pouze nové View, v případě zcela rozdílných vstupních událostí i speciální Controller. Nicméně Model jako klíčová část aplikace zůstává stále stejný.
- Minimalizace duplicitního kódu. Souvisí částečně s předchozím bodem. Například bez oddělení a zapouzdření Modelu by se pro každé nové View musela znovu programovat celá aplikační logika. Ale i v rámci jednoho typu klienta je často jedna akce volána z několika různých míst aplikace.
- Rozdělení vývojářských rolí. Jednotlivé části mohou být vyvíjeny samostatně, jen se znalostí předem určených rozhraní mezi nimi. Minimalizuje se dopad modifikací, změny jsou většinou prováděny jen v dané vrstvě.

Metoda CRUD	HTTP metoda	SQL příkaz
Create	PUT	Insert
Retrieve	GET	Select
Update	POST	Update
Delete	DELETE	Delete

Tabulka 1: Vazby mezi metodami HTTP protokolu, SQL příkazy a jednotlivými CRUD metodami u RESTful aplikace

- Znovupoužitelnost kódu. Jako Model lze ve vlastní aplikaci použít standardní knihovny či třídy. Existují také univerzálně pojaté Controllery.
- Vysoká komplexnost návrhu a snadná budoucí rozšiřitelnost aplikace, efektivní modularita.
- Díky rozčlenění na více logických částí umožňuje aplikaci snáze pokrýt automatizovanými testy.

4.4 REST: Representational State Transfer

REST je architektura rozhraní, navržená pro distribuované prostředí, která byla představena v roce 2000 v disertační práci Roye Fieldinga, jednoho ze spoluautorů protokolu HTTP [10].

REST je, na rozdíl od známějších XML-RPC či SOAP, orientován datově, nikoli procedurálně. Webové služby definují vzdálené procedury a protokol pro jejich volání, REST určuje, jak se přistupuje k datům nebo-li ke zdrojům (resources) [10]. Tento rozdíl se projevuje i v URL. Všechny zdroje mají vlastní identifikátor URI. Zdrojem mohou být data, stejně jako stavy aplikace, pokud je lze popsat konkrétními daty. REST definuje čtyři základní metody pro přístup ke zdrojům [10].

4.4.1 Metody přístupu ke zdrojům

REST umožňuje přistupovat k datům na určitém místě pomocí standardních metod HTTP. Implementuje čtyři základní metody, které jsou známé pod označením CRUD, tedy vytvoření dat (Create), získání požadovaných dat (Retrieve), změnu (Update) a smazání (Delete) [10]. Tyto metody jsou implementovány pomocí odpovídajících metod HTTP protokolu a odpovídajících SQL příkazů znázorněných v tabulce 1.

4.5 RIA: Rich Internet Application

RIA aplikace jsou webové aplikace, které mají charakteristiky desktopových aplikací. Snaží se v rámci webového prohlížeče překlenout rozdíly mezi klasickou webovou aplikací a desktopovou aplikací, a svým vzhledem i chováním a poskytnout vyšší uživatelský komfort [14]. Současné možnosti webových aplikací jsou z hlediska funkcí a uživatelského

komfortu limitovány možnostmi webových prohlížečů. Za RIA aplikaci můžeme považovat takové aplikace, které dokáží splnit ty nejnáročnější požadavky z níže zmíněných oblastí: [14]

- komplexnost grafického rozhraní (MDI koncept)
- uživatelsky přívětivého chování (odstínění od bezstavového protokolu HTTP)
- komfort funkcí odpovídající klasickému desktopovému řešení (drag & drop, klávesové zkratky, kontextová nápověda)

V dnešní době se pro vývoj RIA aplikací nabízí dva způsoby implementace:

- použití proprietární technologie (Flash, Silverlight, Java applety, Java aplikace)
- maximální využití stávajících technologií a prostředků (HTML, CSS, JavaScript)

Problémem druhého způsobu jsou poměrně omezené možnosti prezentačních technologií (HTML, CSS) pro kompaktnější grafická rozhraní. Dále se musí aplikace vyrovnat s faktem, že HTML protokol je bezstavový. Sebemenší změna stavu (autokompletace dat, validace, aktualizace části dat) vyvolaná klientem musí vyvolat také požadavek na server, který jej musí obsloužit a zpět vrátit všechna potřebná data.

4.5.1 AJAX: Asynchronous JavaScript and XML

AJAX není sám o sobě implementací technologie či softwarovým produktem. Jedná se o obecný koncept. Je představitelem cesty tvorby RIA aplikací využívající maximální možné hodnoty stávajících technologií, pod kterými jsou myšleny tyto technologie: [14]

- Document Object Model (DOM)
- XMLHttpRequest (Mozilla, MSIE)
- HTML, CSS a JavaScript

Základním stavebním kamenem je objekt XMLHttpRequest, který umožňuje asynchronní volání serveru. V klasickém webovém modelu každá změna stavu na klientu vyžaduje obnovení celého uživatelského rozhraní. Nejdříve je tedy žádost o změnu stavu, odeslání požadavku na server, vyřízení požadavku a vše končí zasláním kompletního uživatelského rozhraní s daty, přičemž jednotlivé kroky jsou vzájemně synchronizovány. Naopak AJAX, díky XMLHttpRequest, může vyvolat libovolný počet nezávislých požadavků, jejichž výsledky mohou ovlivnit pouze patřičné části uživatelského rozhraní, bez nutnosti jeho celkového znovunačtení. Tedy, žádost o změnu stavu, vygenerování požadavku přes XMLHttpRequest, vyřízení požadavku serverem a zpracování vrácené odpovědi XMLHttpRequestem a změna patřičné části uživatelského rozhraní [14].

5 Analýza a návrh informačního systému

Před návrhem informačního systému bylo nutné se seznámit se současným řešením správy úkolů, používaným skupinou Virtlab.

5.1 Studie současného řešení

Současný informační systém VirtIS, který je používaný skupinou Virtlab ke správě projektů a úkolů, je poměrně obsáhlý a praxí ověřený informační systém. Postupně byly doimplementovávány funkce vyplývající z potřeb týmu.

Konceptuálně není na informačním systému VirtIS nic špatného, je navrhnut a implementován konkrétním potřebám týmu Virtlab. Nepodporuje však základní rysy, které se čekají od aplikace podporující metodiku Getting Things Done. Především aplikace metodiky GTD tak bude mým úkolem při návrhu a implementaci nového informačního systému, který by měl nahradit informační systém VirtIS. Požadavky kladené na aplikaci podporující GTD jsou rozebrány v kapitole 5.2.1.2. Dalšími drobnějšími nedostatky, které se budu snažit vyřešit, jsou uživatelská přívětivost a celková použitelnost informačního systému.

5.2 Specifikace požadavků

Požadavky se vyplatí rozdělit na funkční, které popisují požadovanou funkci systému, a nefunkční, které představují podmínky uvalené na daný systém.

5.2.1 Funkční požadavky

Funkční požadavky popisující požadovanou funkci informačního systému dále rozdělím na:

1. základní obecné požadavky, které by měla podporovat jakákoliv aplikace pro správu úkolů
2. požadavky z hlediska metodiky GTD, které by měla podporovat jakákoliv aplikace využívající metodiku Getting Things Done

5.2.1.1 Základní sada funkcí Na základě konzultace s vedoucím práce a členem týmu skupiny Virtlab byla stanovena následující sada základních funkčních požadavků.

Uživatel se přihlásí do systému. Měl by mít následující možnosti:

- přehledně zobrazit aktuální úkoly
- zpřístupnit pouze uživatelem vytvořené či na uživatele delegované úkoly
- zaměřit svou pozornost na důležité úkoly, které se mají daný den zpracovat

- přidat úkol, vyplnit jeho detaily a označkovat si ho pomocí štítků
  - mezi nejdůležitější detaily úkolů patří název úkolu, poznámka, termín splnění, priorita, informace o delegaci na jiného uživatele, odhadovaný a reálný čas
- vyhledávat úkoly především podle jména, štítků a termínu splnění
- delegovat úkoly na jiné uživatele systému
  - uživatel, na kterého byl úkol delegován, musí být upozorněn e-mailem
  - zadavatel úkolu může být i jeho řešitelem (v tomto případě neupozorňovat e-mailem)
  - je-li delegovaný úkol hotov, měl by jej uživatel, který jej delegoval zkontrolovat a odsouhlasit
- jednoduše manipulovat s úkoly
  - delegace, odložení, přesun mezi projekty a seznamy pomocí *drag & drop* principu
- zobrazit týdenní zhodnocení
  - počet odpracovaných hodin
  - počet k hodin k proplacení
- seskupovat úkoly do projektů
- nastavit projektu údaj o hodinovém rozpočtu a ceně za hodinu práce
- nastavit projektu příznak, zda-li se jedná o aktivní či archivovaný projekt
- uspořádávat úkoly podle pořadí vykonávání
- export úkolů v otevřených formátech pro výměnu kalendářových dat, např. iCalendar

5.2.1.2 Požadavky z hlediska aplikace metodiky Getting Things Done

Má-li aplikace splňovat požadavky metodiky Getting Things Done, musí splňovat následující kritéria vyplývající z rozboru metodiky z kapitoly 3.1.

- Systém musí zajistit kontrolu nad stovkami nových vstupů denně.
- Systém musí ušetřit mnohem více času a úsilí, než kolik je ho třeba k jeho udržování.
- Systém musí umožnit identifikovat a organizovat další kroky projektů a úkoly, které na něco čekají (delegované úkoly).

- Jakmile je úkol delegován, zmizí z dohledu.
- Vedle seznamu úkolů je třeba mít k dispozici i seznam spolupracovníků, pokud v rámci jednoho projektu jsou delegovány úkoly i na jiné uživatele.
- U každého spolupracovníka by mělo být možné zobrazit, jaké úkoly na něj byly delegovány a v jakém jsou stavu.
- Je doporučeno systém doplnit o čtyři úrovně priorit dle GTD (*můžu, chtěl bych, měl bych, musím*).

5.2.2 Nefunkční požadavky

Na aplikaci nebyly kladeny žádné funkční požadavky ani omezení z hlediska implementačního jazyka či SŘBD. Bylo tedy třeba zvolit implementační prostředí, vhodný framework i SŘBD. Výběrem vhodného implementačního prostředí se zabývá kapitola 6.1

5.3 Datová analýza

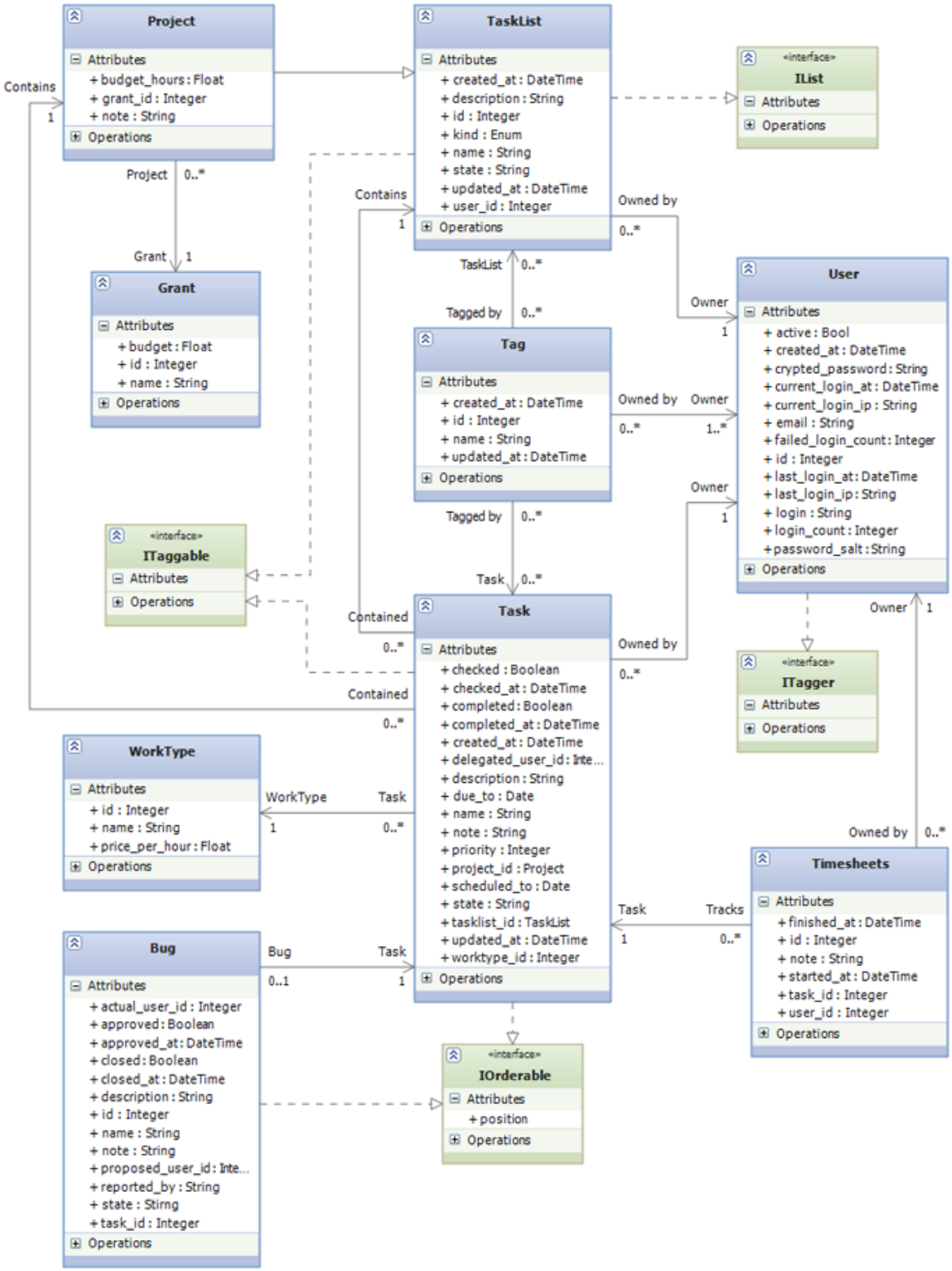
5.3.1 Konceptuální model

Jak vyplývá z požadavků definovaných v kapitole, je třeba vytvořit systém pro správu úkolů a projektů s možností hlášení chyb a zaznamenávání času stráveného nad jednotlivými úkoly.

V následujících podkapitolách jsou slovně popsány potřeby jednotlivých tříd, které budou v rámci modelovaného systému vystupovat na logické úrovni. Výstup je pak zaznamenán v třídním UML diagramu popisující konceptuální model na obrázku 3. Pro vizualizaci konceptuálního modelu byl upřednostněn jazyk UML před ERD, především kvůli jeho standardizaci, možnostem a rozšířenosti. Právě pro jeho široké možnosti modelování bude i dále v analýze využito převážně tohoto grafického jazyka.

5.3.1.1 Třída User Třída User reprezentuje uživatele systému, u kterého je třeba evidovat následující atributy:

- login a e-mail uživatele
- heslo v zakódované podobě
- počet přihlášení do systému
- počet neúspěšných přihlášení do systému
- současnou IP adresu, ze které je uživatel přihlášen
- poslední IP adresu, ze které byl uživatel přihlášen
- datum a čas vytvoření uživatele, posledního a předposledního přihlášení
- příznak aktivace či deaktivace účtu



Obrázek 3: Konceptuální datový model modelovaného systému zobrazený pomocí třídního UML diagramu

5.3.1.2 Třída TaskList Třída TaskList reprezentuje seznam úkolů. Je třeba u něj evidovat tyto atributy:

- unikátní identifikátor
- název a popis seznamu úkolů
- čas a datum vytvoření a poslední aktualizace
- typ seznamu (*I*, *N*, *S*)

5.3.1.3 Třída Project Třída Project je generalizací třídy TaskList. Obsahuje tedy všechny atributy, které jsou definovány ve třídě TaskList, a navíc je potřeba u ní evidovat atributy:

- unikátní identifikátor
- hodinový rozpočet
- poznámky k projektu
- vlastníka projektu
- zdroj financování

5.3.1.4 Třída Task Další z klíčových tříd modelovaného systému je třída Task. U ní je třeba evidovat tyto atributy:

- unikátní identifikátor
- název
- vlastníka a případného řešitele úkolu (pokud se liší)
- případný podrobnější popis a poznámku řešitele
- stav úkolu
  - hotový (*completed*) - zadává řešitel úkolu
  - přiřazený (*delegated*)
  - zkontrolovaný (*checked*) - zadává zadavatel úkolu, jde-li o delegovaný úkol
- datum a čas vytvoření, poslední editace, vyhotovení a zkontrolování úkolu
- datum, do kdy musí být úkol splněn
- den, na který je úkol naplánován k řešení
- hodinovou sazbu
- seznam, kterého je úkol součástí (volitelně)
- projekt, kterého je úkol součástí (volitelně)

5.3.1.5 Třída Bug Doplňující třídou reprezentující hlášené chyby do systému je třída Bug. U ní je třeba evidovat tyto atributy:

- unikátní identifikátor
- popis chyby
- iniciály uživatele, který chybu nahlásil
- navrhovaný řešitel
- skutečný řešitel
- stav
  - schválený (*approved*)
  - uzavřený (*closed*)
- datum a čas, kdy došlo k schválení chyby a k jejímu uzavření
- reference na případný úkol, který byl z Bugu vygenerován a který se k němu váže

5.3.1.6 Třída Timesheet Třída Timesheet představuje časový záznam, který bylo potřeba vytvořit ke splnění úkolu. U této třídy je třeba evidovat následující atributy:

- unikátní identifikátor
- úkol, kterého se týká a jeho řešitele
- datum a čas, kdy začal a kdy skončil s prací
- případnou poznámku řešitele

5.3.1.7 Třída Tag Třída Tag představuje štítek, kterým mohou být označeny projekty a úkoly pro jednodušší vyhledávání. U této třídy je třeba evidovat jen tyto atributy:

- unikátní identifikátor
- název štítku
- volitelně datum i čas vytvoření a poslední úpravy

5.3.1.8 Číselníky Aplikace se taktéž neobejde bez číselníků. Mezi dva nejvýznamnější patří číselník *prací a sazeb* a číselník *zdrojů a financování*. U obou je třeba evidovat následující atributy:

- unikátní identifikátor
- název grantu či typu práce
- sazba

5.3.2 Datový model

Datový model modeluje realitu do databáze podle typu použitého SŘBD. Datový slovník je kvůli rozsáhlosti uveden v příloze.

5.4 Analýza procesů

Jelikož konceptuální modelování poskytuje pouze statický pohled na navrhovaný systém, identifikuji a definuji v této podkapitole jednotlivé procesy systému, které jsou požadovány jednotlivými typy uživatelů.

5.4.1 Identifikace procesů

K identifikaci procesů se nejčastěji používá grafické znázornění *Use case* modelů.

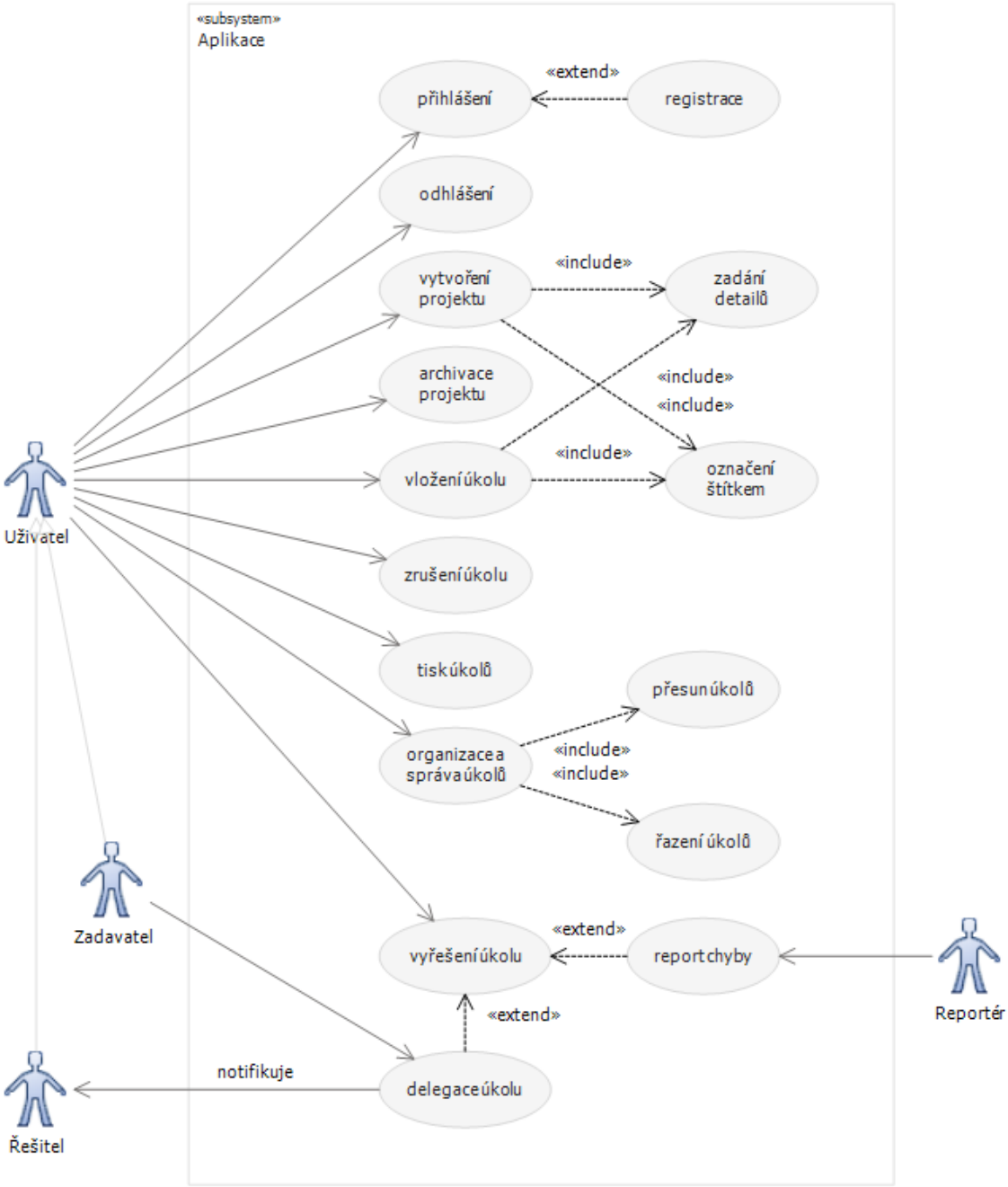
Use case model je zobrazení dynamické funkční struktury systému z pohledu různých typů uživatelů. Je určen k definici chování systému, aniž by odhaloval jeho vnitřní strukturu. Chování systému definuje jako soubor scénářů, kdy každý scénář obsahuje sekvenci událostí, které v jeho rámci probíhají, a popis interakce mezi uživatelem a systémem.

Use case model identifikující procesy v budovaném systému pro správu projektů a úkolů je zobrazen na obrázku 8.

5.4.2 Seznam procesů

Nyní, po identifikaci procesů, můžu zhotovit jejich kompletní seznam a zapsat je hierarchicky do logických celků. Některé triviální procesy, jako operace CRUD nejsou z důvodu úspornosti a předcházení efektu *paralýzy analýzou* obsaženy.

1. Evidence uživatelů; tabulka `users`
   (a) Aktivace uživatele
   (b) Deaktivace uživatele
   (c) Přihlášení uživatele
   (d) Odhlášení uživatele
2. Evidence seznamu úkolů; tabulka `tasklists`
   (a) Přidání úkolu
3. Evidence projektů; tabulka `projects`
   (a) Přidání úkolu
   (b) Archivace projektu
4. Evidence úkolů; tabulka `tasks`
   (a) Delegace úkolu



Obrázek 4: Identifikace procesů pomocí Use case modelu

(b) Přidání úkolu do existujícího projektu nebo seznamu úkolů
(c) Změna pořadí úkolů v rámci existujícího projektu nebo seznamu úkolů
(d) Obnovit smazaný úkol
(e) Vyřešení úkolu

5. Evidence chyb; tabulka `bugs`
   (a) Schválení chyby (označení k řešení)
   (b) Vytvoření úkolu k opravení chyby
   (c) Změna pořadí chyby v rámci seznamu chyb
   (d) Uzavření chyby
6. Evidence prací na úkolech; tabulka `timesheets`
   (a) Pouze základní CRUD operace
7. Evidence štítků; tabulka `tags`
   (a) Přidání štítku k existujícímu projektu
   (b) Přidání štítku k existujícímu úkolu
   (c) Odebrání štítku z existujícího projektu
   (d) Odebrání štítku z existujícího úkolu

5.4.3 Podrobný popis procesů

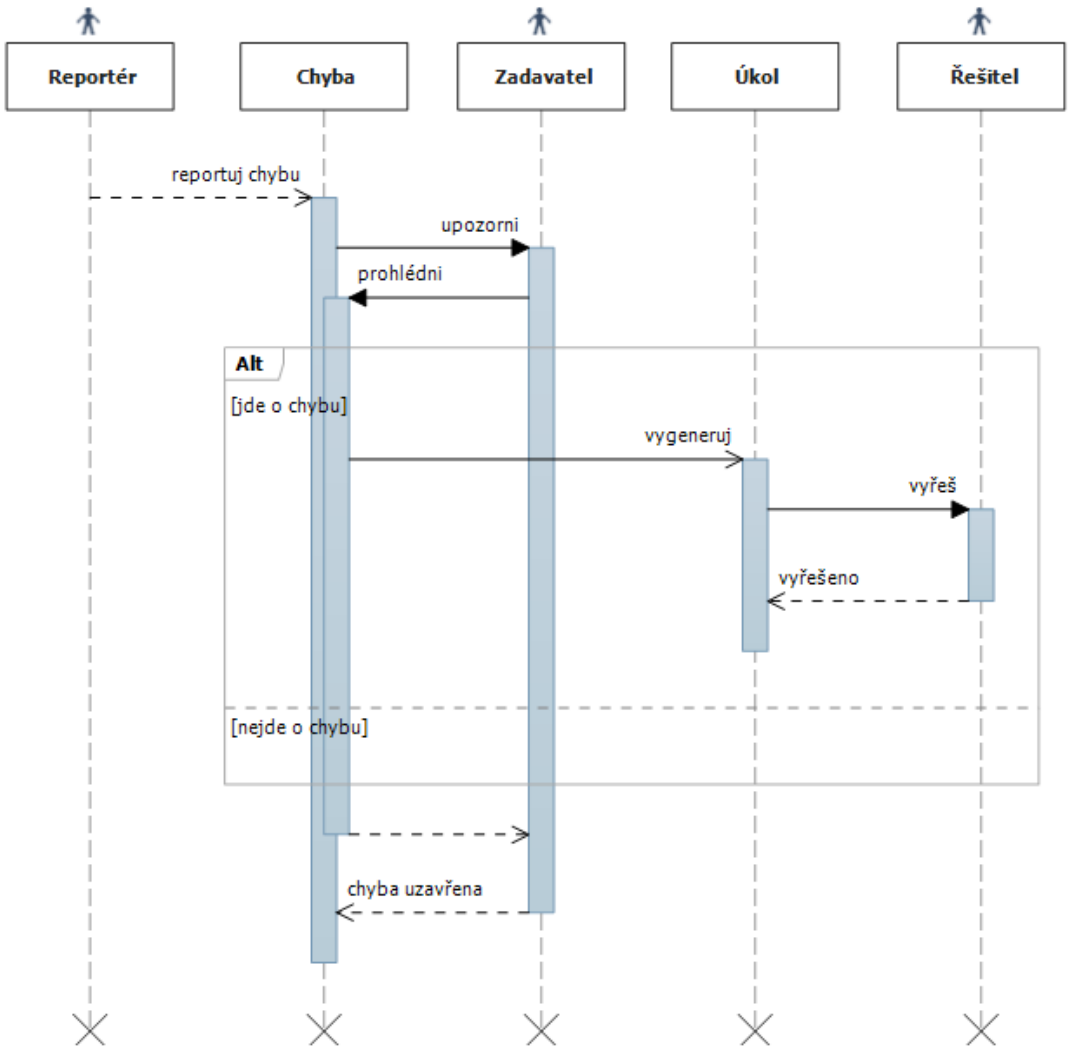
Seznam procesů obsahuje z větší části triviální procesy, které se skládají maximálně z několika málo kroků. Jediným významnějším a složitějším procesem je proces delegace úkolu, zakládá-li se úkol na reportované chybě.

5.4.3.1 Delegace úkolu založeném na reportované chybě Celý proces začíná reportováním chyby *reportérem*. *Zadavatel* poté deleguje úkol *řešiteli*, který je vyřeší či jinak zpracuje. Celý tento proces symbolizuje bug trackingový systém[3] podobný tomu, jak je použitý v aplikaci VirtIS. Celý tento proces je znázorněn v sekvenčním diagramu na obrázku 5.

5.5 Dynamická analýza

Mezi objekty s komplikovanější logikou stavů v budovaném systému patří především objekty úkolu a chyby. Je proto třeba popsat a definovat modely životního cyklu těchto objektů, jednotlivé vztahy a přechody mezi nimi. Dynamická analýza popisuje časová následnost operací, životní cyklus entit a stavy databáze a IS.

[3] Bug trackingový systém je aplikace pro sledování a hlášení softwarových chyb.



Obrázek 5: Sekvenční diagram procesu delegace úkolu založeném na reportované chybě

V následujících odstavcích budu používat označení pro různé uživatelské role. Jde o uživatelské role definované v identifikaci procesu z Use case modelu na obrázku 8.

Uživatelské role jsou tedy následující:

- *reportér* reportuje chyby do Bug trackingového systému
- *zadavatel* zadává nebo-li deleguje úkoly řešitelům
- *řešitel* řeší přiřazené úkoly
- *uživatel* je generalizovaná role, pracuje se systémem a má možnosti jak *zadavatele*, tak *řešitele*

5.5.1 Seznamy úkolů a logika přesunů úkolů mezi nimi

V rámci aplikace metodiky GTD je třeba do systému zavést a rozlišovat následující typy seznamů[4]:

- Vstupní schránka (*Inbox*)
- Seznam dalších kroků (*Next actions*)
- Seznam úkolů, které je potřeba vykonat dnes (*Today*)
- Seznam úkolů, které je potřeba následující den (*Tomorrow*)
- Kalendář (*Scheduled*)
- Seznam úkolů někdy – možná (*Someday*)
- Seznam delegovaných úkolů (*Delegated*)
- Seznam hotových úkolů (*Completed*)
- Seznam smazaných úkolů (*Trashed list*)

Seznamy *Inbox, Next actions a Someday* jsou jediné typy seznamů, které jsou uložené v databázi. Platí pro ně některá pravidla:

- Úkol může být přesouván v rámci těchto seznamů, vždy ale může být jen na jednom z nich současně.
- Úkol je ihned po vytvoření automaticky obsažen v seznamu *Inbox*.
- Seznam *Inbox* obsahuje všechny nehotové a na uživatele delegované úkoly, které nebyly přesunuty do seznamu *Next actions* nebo *Someday*.
- Do seznamů *Next actions a Someday* může být úkol manuálně přesunut v rámci fáze zpracování.

[4]Pro označení typů seznamů budou dále používány anglické názvy uvedené kurzívou.

- Úkol obsažen v seznamu *Inbox* nebo v seznamu *Someday* se nemůže zobrazovat u žádného jiného seznamu, může být ale obsažen v projektu nebo v oblasti zodpovědnosti.
- Úkol obsažen v seznamu *Next actions* se může zobrazovat v seznamech *Today, Tomorrow, Scheduled*, nebo být obsažen v projektu či v oblasti zodpovědnosti.

Seznamy *Today, Tomorrow, Scheduled, Delegated, Completed a Trashed* jsou virtuální seznamy úkolů, které jsou vytvořeny na základě splnění některých podmínek:

- Seznam *Tomorrow* zobrazuje nehotové úkoly s vyplněným atributem `scheduled_to` s hodnotou rovnu datumu následujícího kalendářního dne.
- Úkolu přesunutému na seznam *Tomorrow* se nastaví hodnota atributu `scheduled_to` na datum následujícího kalendářního dne.
- Seznam *Scheduled* zobrazuje nehotové úkoly s vyplněným atributem `scheduled_to`, které ale současně nejsou obsaženy v seznamu *Today* a *Tomorrow*.
- Jakmile je současné kalendářní datum shodné nebo větší než hodnota atributu `scheduled_to`, začne se úkol zobrazovat v seznamu *Today*.
- Seznam *Delegated* zobrazuje nehotové úkoly, které uživatel delegoval na některého z jiných uživatelů.
- Seznam *Completed* zobrazuje pouze hotové úkoly.
- Seznam *Trashed* zobrazuje pouze úkoly, které byly označeny ke smazání.

Dalšími rozšiřujícími typy seznamů jsou projekty a oblasti zodpovědnosti. Zde je jen jedno pravidlo:

- Jakýkoliv úkol může být vždy právě jen v jednom projektu nebo oblasti zodpovědnosti.

5.5.2 Životní cyklus úkolu

1. úkol je vytvořen *uživatelem* či vygenerován z *reportérem* nahlášené chyby
2. úkolu jsou vyplněny detaily
3. úkol je přesunován mezi seznamy úkolů nebo projekty
4. úkol je označen jako **hotový**, smazán nebo delegován na některého z *řešitelů*

5.5.3 Životní cyklus chyby

1. chyba je nahlášena (vytvořena formulářem)
2. všem *zadavatelům* se zobrazuje v Inboxu
3. jeden ze *zadavatelů* navrhne *řešitele*
4. jeden ze *zadavatelů* schválí navrhovaného *řešitele*
5. jeden ze *zadavatelů* schválí či zamítne řešení chyby, úkol je **schválený**
   (a) z chyby je vygenerován úkol a delegován na *řešitele*, úkol je **přiřazený**
   (b) z navrhovaného *řešitele* se stává skutečný *řešitel*
   (c) *zadavatel* tento úkol přiřadí do některého ze seznamu úkolů nebo projektu, vyplní název a dodatečné informace, čas splnění, štítky, aj.
   (d) tento úkol se zobrazí v Inboxu skutečného *řešitele*, ten může v úkolu upravovat jen poznámky a přesouvat si jej do svých seznamů úkolů nebo projektů
   (e) dále *řešitel* pracuje s úkolem tak, jak je popsáno v životním cyklu úkolu
   (f) jakmile je úkol *řešitelem* označen jako **hotový**, je předán *zadavateli* ke kontrole
   (g) *zadavatel* vidí, že byl úkol splněn, zkontroluje správnost řešení, je-li s řešením
      - nespokojen, neschválí korektnost řešení a *řešiteli* se tento úkol opět zobrazí v seznamech úkolů
      - spokojen, odsouhlasí správnost řešení a úkol je označen jako **zkontrolovaný** a chyba, ze které byl úkol vygenerován, je uzavřena

5.6 Výstup analýzy

Výstupem analýzy je celkový pohled na modelovaný systém. Jsou známy všechny třídy, jejich atributy, metody a vzájemné vazby mezi třídami. Díky tomu můžeme vizualizovat tento výstup do třídního diagramu[5] jazyka UML, který je zobrazen na obrázku 6.

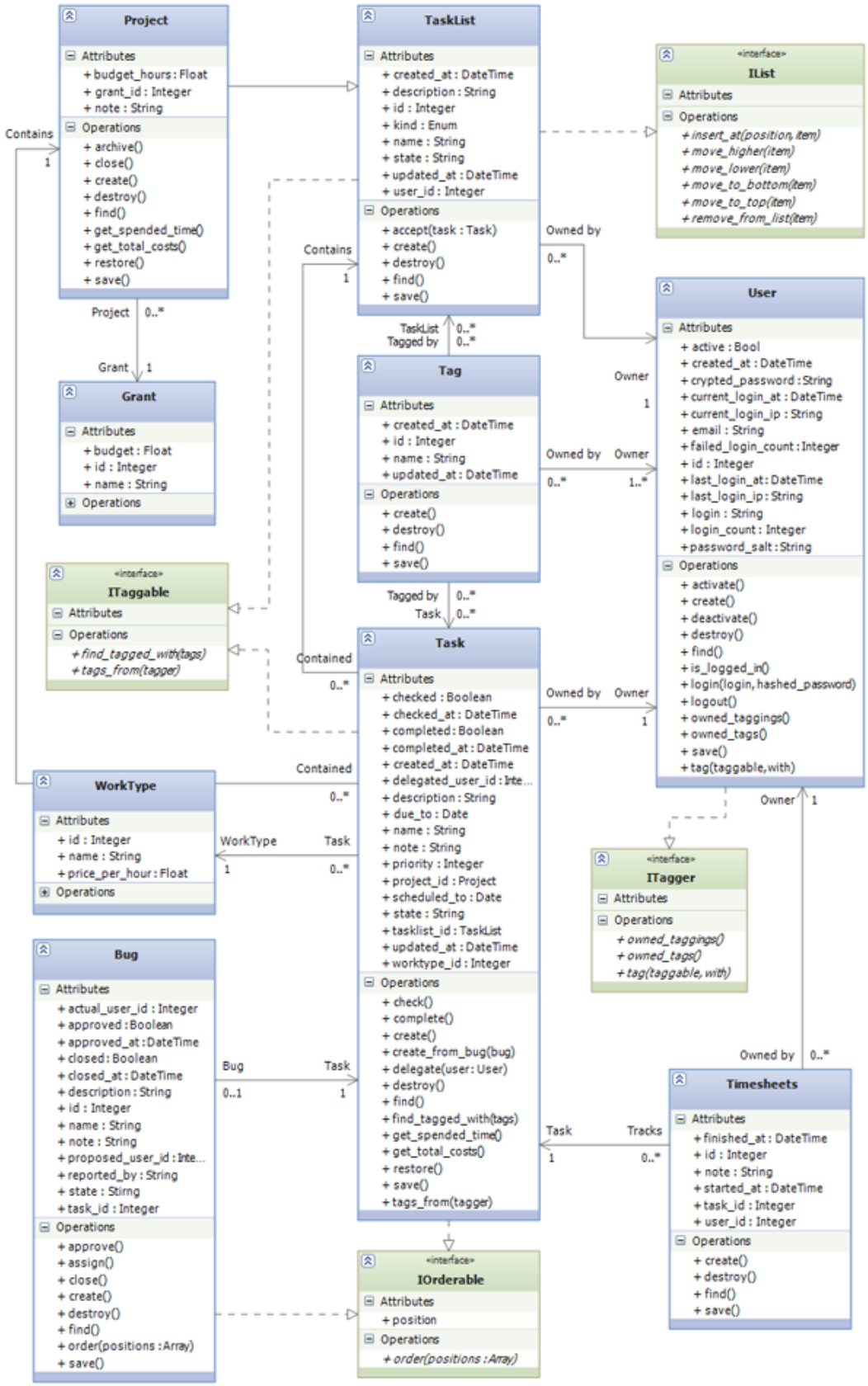
5.7 Návrh uživatelské rozhraní

Současný informační systém používaný skupinou Virtlab ke správě projektů a úkolů je poměrně obsáhlý na počty vstupů a formulářových prvků. Tato obsáhlost možná vede k jisté míře nepřehlednosti pro uživatele, který se s ním setkává poprvé.

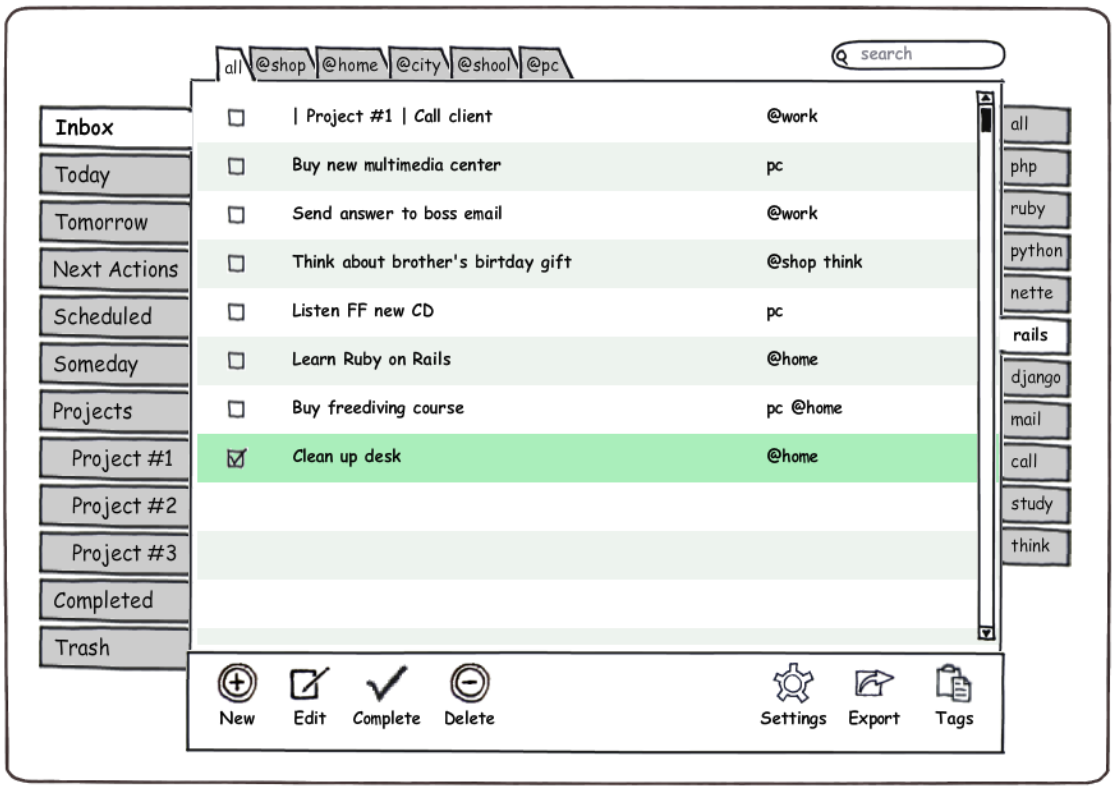
Právě rozhraní aplikace často rozhoduje, zda-li bude aplikace úspěšná či ne. V oblasti UI stávající aplikace se nabízí velký prostor pro další vylepšení.

Například zadávání nových úkolů by mělo být jednoduché a musí minimalizovat uživatelovo úsilí při vkládání velkého počtu úkolů. V systému VirtIS obsahuje formulář pro přidání úkolů až zbytečně velké množství vstupů o detailech, které uživatel při vytváření úkolů nejspíše ani nezná. Je proto důležité se v této fázi zamyslet nad tím, kterými

[5] Všechny třídy zobrazené na třídním diagramu mají imlicitně stereotyp `persistent`.



Obrázek 6: Výsledný třídní diagram modelované aplikace



Obrázek 7: Prototyp uživatelského rozhraní aplikace

detaily uživatele zatěžovat a které jsou pro něj naopak nepodstatné. Mezi tyto detaily například určitě nepatří poznámka realizátora, která se vyplňuje až při uzavírání úkolu.

Další částí informačního systému, o které se domnívám, že by stála za přepracování, jsou výpisy seznamů úkolů. Výpisy nejsou nijak stránkovány, takže při velkém počtu úkolů vznikají velmi dlouhé seznamy. Jedním z řešení je zobrazovat méně detailů ve výpisech a tyto podrobnosti zobrazovat až v detailech úkolů. Dalším z možných řešení je rozšíření o stránkování nebo také kombinace obou řešení.

Na trhu dnes existuje několik velmi oblíbených a povedených aplikací podporující metodiku GTD. Liší se především ve způsobu implementace. Některé jsou implementovány jako webové aplikace, jiné zase jako desktopové aplikace využívající moderních technologií jako Adobe AIR nebo Microsoft Silverlight. V základní funkčnosti se většinou shodují stejně tak jako ve filozofii pojetí metodiky GTD a způsobu implementace všech seznamů důležitých právě z hlediska GTD, jako je seznam dalších kroků, schránka na došlé záležitosti, seznam úkolů pro dnešní den či způsob označování úkolů štítky a kontexty.

Typický prototyp[6] těchto aplikací je znázorněn[7] na obrázku 7. Jedním z cílů implementace bude právě dodržet podobně jednoduché rozhraní aplikace, aby se zachovala logika a částečně i vzhled rozhraní pro případný přechod uživatelů na jiný nástroj podporující metodiku GTD.

Mezi nejpopulárnější nástroj podporující metodiku GTD patří program *OmniFocus*[8], který je doporučován samotným autorem metodiky Getting Things Done. Dalším velmi populárním nástrojem je program *Things*[9] vyvíjený německou firmou *Cultured Code*. Ani jeden z těchto programů bohužel není multiplatformní. Pro jejich použití je vyžadován operační systém Mac OS X a oba mají i svou mobilní verzi pro platformu iPhone OS. Dalšími populárními nástroji jsou webová aplikace *Remember The Milk*[10] a desktopová aplikace *Doit.im*[11]. Výhodou těchto nástrojů je především nezávislost na platformě a díky mobilním klientům také přístupnost odkudkoliv.

[6] Prototypování je jedním ze současně populárních nástrojů pro tvorbu uživatelského rozhraní.

[7] Prototyp nebo-li *wireframe* byl vytvořen pomocí programu Balsamiq Mockups volně dostupného z http://www.balsamiq.com

[8] http://www.omnigroup.com/products/omnifocus/

[9] http://culturedcode.com/things/

[10] http://www.rememberthemilk.com/

[11] http://www.doit.im/

6 Implementace informačního systému Planner

Implementace informačního systému je promítnutí analýzy do implementačního prostředí. V této kapitole provedu výběr implementačního prostředí, aplikačního frameworku a popíši jeho vlastnosti, přednosti a způsob práce.

6.1 Výběr vhodného aplikačního frameworku

Pro implementaci informačního systému byl vybrán webový aplikační framework Ruby on Rails pro jazyk Ruby. Výběr probíhal mezi ním a jazykem PHP v kombinaci s Nette Frameworkem a databází PostgreSQL. Výběr proběhl na základě napsání jednoduché aplikace, která měla ukázat přednosti obou řešení. Rozhodnutí tedy proběhlo na základě subjektivního dojmu především z jednotlivých aplikačních frameworků.

Jazyk Ruby a framework Ruby on Rails se v praxi ukázal jako vhodný pro jakýkoliv typ webové aplikace, ať už jde o jednoduché webové prezentace, či velmi populární služby jako software pro řízení spolupráce Basecamp, sociální síť Twitter, platformu internetového podnikání a elektronických obchodů Shopify, systém pro správu chyb Lighthouse, management komunitního vývoje a hostingu (nejen) open source projektů Github či jiné webové aplikace[12]. Framework má za sebou již dlouhý vývoj, je tedy vyzrálý a do jeho údržby a vývoje je zapojeno přes 1500 vývojářů[13]. Zároveň také splňuje všechny funkční požadavky pro pokročilejší webové aplikace, které byly zmíněny v kapitole 4.2.

Mezi dalšími pozitivními faktory, díky kterým bylo rozhodnuto k použití frameworku Ruby on Rails byly:

- jazyk Ruby je plně objektový, zdrojové kódy jsou velmi úsporné a přesto velmi dobře čitelné, dokonce i nepříliš zběhlým programátorům
- velké množství rozšíření a doplňků frameworku i jazyka
- množství kvalitních a dostupných výukových materiálů a s tím související strmá křivka učení frameworku
- důsledně dodržuje Model-View-Controller architekturu
- pracuje s většinou dnes rozšířených relačních databází
- databázová struktura se udržuje přímo v aplikaci pomocí tzv. *migrací*, díky kterým je možno kdykoliv v průběhu zaměnit SŘBD či používat různé SŘBD pro vývojové a pro produkční prostředí
- pro pohodlný přístup k datům používá objektově-relační mapování (ORM)
- jednoduchá tvorba RESTful aplikací díky přítomnosti promyšleného způsobu routování

[12] Dle oficiálních stránek frameworku, http://rubyonrails.org, duben 2010

[13] Údaj pochází ze seznamu přispěvatelů na vývoji platnému k dubnu 2010 uveřejněném na adrese http://contributors.rubyonrails.org

- podpora pro tvorbu RIA aplikací díky integraci JavaScriptového frameworku Prototype a jeho rozšíření Scriptaculous pro práci s UI, funkční ve všech běžných prohlížečích
- velmi dobrá podpora jazyka i frameworku v integrovaných vývojových prostředích
- je navržen pro agilní vývoj webových aplikací s důrazem na vysokou produktivitu programátorských týmů
- výsledné aplikace jsou dobře testovatelné vývojovou technikou Test-driven development (TDD)

6.2 Agilní vývoj

V posledních letech se v rámci menších vývojových týmů začíná často prosazovat tzv. agilní technika vývoje aplikací. Jedná se o souhrn principů a metod, jak co nejrychleji a nejefektivněji vyvinout software tak, aby odpovídal požadavkům zákazníka. Cílem této techniky je zvýšení produktivity práce, odstranění chyb, jejich předcházení a kvalitnější software.

Základní rozdíl od mnoha pouček o správném vzniku software lze spatřit ve zkrácení délky intervalu návrh – implementace – testování – oprava, a zvýšení počtu jeho iterací. Oproti klasickému vývoji dochází většinou k úpravám či rozšíření pouze malé části zdrojového kódu dle změn požadavků zákazníka. Manifest doporučuje dodávat fungující části software k posouzení v co nejkratších intervalech (14 dní až 2 měsíce). Pro tento způsob vývoje se také používá označení *inkrementální programování* [9].

Agilní techniky snižují význam analýzy a je zvykem, že při vývoji se téměř neprodukuje dokumentace v tradičním slova smyslu. Tu zastupuje vlastní zdrojový kód, který má být strukturovaný, čitelný a správně dokumentovaný. Návrh aplikace již nepředstavuje etapu vývoje dokončenou před započetím implementace, ale stává se jeho součástí. Změny v průběhu práce jsou vítány a chápány jako pozitivní vývoj událostí [9].

Vybraný aplikační framework Ruby on Rails splňuje všechny nutné předpoklady k agilnímu vývoji aplikace. Dá se říct, že framework je této metodice přímo navržený a vývojáři toho hojně využívají. Implementace aplikace se tedy nese v duchu agilního vývoje. Důraz je kladen zejména na univerzálnost a rozšířitelnost aplikace.

6.3 Ruby on Rails

Ruby on Rails je opensource platforma postavená na dynamickém skriptovacím jazyce Ruby. Je navržena pro agilní vývoj webových aplikací s důrazem na vysokou produktivitu programátorských týmů. Ruby on Rails důsledně dodržuje Model-View-Controller architekturu, která umožňuje jednoduché a spolehlivé oddělení business logiky od prezentační vrstvy. Pracuje s většinou dnes rozšířených relačních databází (včetně Oracle a MS SQL server). Pro pohodlný přístup k datům používá objektově-relační mapování (ORM) [8].

6.3.1 Vlastnosti frameworku

Hlavní výhodou frameworku Ruby on Rails je vysoká úspornost a flexibilita kódu, především díky vlastnostem jazyka Ruby. Pomocí poměrně málo příkazů je možné definovat rozsáhlou funkcionalitu. Tato úspornost následně umožňuje vysokou rychlost vývoje aplikací, jednodušší následné změny a lepší udržovatelnost aplikací [8]. Framework Ruby on Rails je zaměřen velice pragmaticky. Neobsahuje žádné zbytečnosti nebo uměle vynalezené technologie.

Framework Ruby on Rails byl extrahován z několika dobře fungujících *Web 2.0* aplikací. Obsahuje kolekci doporučených postupů týkajících se vývoje webových aplikací. Jeho autory jsou programátoři z praxe, kteří jej napsali proto, aby jim ulehčil a zefektivnil každodenní práci. Ruby on Rails je snadné se naučit a díky vlastnostem jazyka Ruby se dobře rozšiřuje o další moduly [8].

V následujících podkapitolách se budu zabývat těmi nejdůležitějšími moduly a vlastnostmi frameworku. Ukázky zdrojových kódů použité pro demonstraci těchto vlastností nemusí pro zjednodušení přesně odpovídat zdrojovým kódům použitým v implementaci informačního systému, který je součástí této práce.

6.3.2 Objektově-relační mapování

Objektově-relační mapování slouží ke snadnému použití relačních databází v prostředí objektově orientovaných programovacích jazyků. Vzhledem k tomu, že objektově orientovaný návrh dat není jednoznačně převoditelný na relační databáze a opačně, používají se různé formy mapování. Mapování má za účel načítat data z relační databáze a naplnit jimi příslušné datové položky objektů, případně naopak datové položky objektů ukládat do databáze. Snahou ORM je co nejlepší využití obou zmíněných technologií. Objekty by měly reprezentovat objekty reálného světa, jak to požadují principy OOP, na straně databáze bychom zase měli využít všech možností relačních databází, tzn. indexů, pohledů, primárních klíčů aj.

Použití ORM tedy přináší řadu výhod, mezi ty nejpodstatnější patří:

- ulehčuje práci s databází
- odstiňuje programátora od SQL
- umožňuje definovat vztahy mezi tabulkami
- obsahuje výkonné nástroje pro práci s daty (čtení, zápis, validace)

Výběr vhodného ORM framework Ruby on Rails usnadňuje, jelikož je standardně dodáván s knihovnou Active Record, která zajišťuje objektově-relační mapování.

6.3.2.1 Active Record Návrhový vzor Active Record staví na předpokladu, že základem aplikace je databázový model a od něj se vše odvíjí. Každý řádek databázové tabulky nebo pohledu je reprezentován konkrétní instancí objektu, která zajišťuje jeho persistenci a definuje doménovou logiku [15, 7].

Ve frameworku Ruby on Rails je vzor Active Record implementován stejně pojmenovanou knihovnou pro objektově-relační mapování. Instance objektů rozšiřují bázovou třídu. Tato třída implementuje potřebné chování vzoru Active Record a obsahuje prostředky pro definici doménové logiky. Tyto objekty jsou pak nosičem dat a zapouzdřují také metody pro persistenci objektu. Na následujícím výpisu kódu se jedná o řádky 11 až 13. Na řádku 2 jsou volitelně definovány atributy třídy[14]. Doménová logika je definována právě v definici třídy na řádcích 5 a 6. Na řádku 8 jsou pak definovány asociace s dalšími třídami.

```
1  # /app/models/tag.rb
2  class Tag < ActiveRecord::Base
3      attr_accessible  :name
4
5      validates_presence_of    :name
6      validates_uniqueness_of :name
7
8      has_many :users, :projects,  :tasks
9  end
10
11 tag = Tag.find(1)
12 tag.name = 'Ruby on Rails'
13 tag.save
```

Výpis 1: Příklad použití ORM Active Record ve frameworku Ruby on Rails

Tento vzor se hojně využívá v různých ORM frameworcích. Je součástí dnes velmi moderních frameworků jako jsou Ruby on Rails, Django pro Python nebo projekt Castle pro .NET, které na něm stavějí svůj datový model a díky velice snadnému mapování umožní napsat kostru databázové stránky aplikace během několika málo minut. Je třeba ale znát několik pravidel a konvencí:

- konvence má přednost před konfigurací
- název tabulky množné číslo
- název třídy modelu (v ORM) jednotné číslo
- není potřeba definovat v kódu atributy modelů, aplikace si je dokáže zjistit sama, je to však doporučeno pro předcházení bezpečnostnímu riziku hromadného přiřazení

Díky principu konvence před konfigurací není potřeba zdlouhavě konfigurovat nastavení ORM nebo atributy jednotlivých modelů. Odpadají tak nepříjemnosti s množstvím nepřehledných XML, jako například u ORM Hibernate z Javy. Jediné, co programátor konfiguruje, jsou pak přihlašovací údaje k databázi pro různá prostředí (vývojové, produkční a testovací) ve vyhrazeném konfiguračním souboru ve formátu yaml.

[14]Knihovna Active Record nespecifikuje atributy přímo v třídách modelů, jejich definice je přímo v databázi, která se vytváří nejlépe použitím databázových migrací.

6.3.3 Databázové migrace

Pro odstínění od nízkoúrovňových DDL příkazů jsou mnozí vývojáři zvyklí používat interaktivní nástroje k vytváření a udržování databázových schémat. Na první pohled je tento přístup pohodlný a přehledný. Jakýkoliv zásah do databázového schématu je pak ale nezvratný a vývojář za tento komfort ztrátou historie změn, což také činí nasazení vývojové aplikace do produkčního prostředí obtížnějším. Framework Ruby On Rails řeší tento problém nasazením tzv. databázových migrací.

Ve frameworku Ruby on Rails se k definici databázové struktury nepoužívají nízkoúrovňové DLL příkazy (Data Definition Language statements) jako `CREATE TABLE` aj. Místo toho se používá vysokoúrovňový přístup pomocí databázových migrací. Každá migrace reprezentuje změnu, která se bude provádět s databází, vyjádřenou v databázově nezávislém jazyku. Změny se mohou vztahovat jak na strukturu, tak na obsah databáze. Migrace by měla obsahovat dvě definice: jedna aplikující změny migrace na databázi, druhou beroucí tyto změny zpět. Mezi jednotlivými migracemi lze díky tomu bez problémů přecházet. Jednoduchý příklad databázové migrace je na výpise zdrojového kódu 2.

```
1  # /db/migrate/20080823223136_create_tasks.rb
2  class CreateTasks < ActiveRecord::Migration
3    def self.up
4      create_table :tasks do |t|
5        t.integer :project_id
6        t.string :name
7        t.boolean :complete, :default => false, :null => false
8        t.timestamps
9      end
10
11     add_index :tasks, :project_id
12   end
13
14   def self.down
15     drop_table :tasks
16   end
17 end
```

Výpis 2: Příklad databázové migrace ve frameworku Ruby on Rails

6.3.4 Bezpečnost aplikace

U informačních systémů implementovaných jako webové aplikace jsou kladeny vysoké bezpečnostní nároky. Jakákoliv webová aplikace implementovaná ve frameworku Ruby on Rails poskytuje základní bezpečnostní ochranu a zabezpečuje nejrozšířenější bezpečnostní trhliny:

- filtrování citlivých parametrů v logu
- SQL injection

- Session Hijacking
- Cross-Site Request Forgery (CSRF)
- Cross-Site Scripting (XSS)

Sadu konkrétních postupů, jak aplikaci zabezpečit, lze nalézt v on-line dokumentaci frameworku[15].

6.3.5 Použití doplňků třetích stran

Jazyk Ruby má spoustu pomocných nástrojů, jako je například balíčkovací systém RubyGems nebo buildovací systém Rake.

Balíčkovacím systémem RubyGems můžeme instalovat rozšíření a doplňky frameworku Ruby on Rails, ale i rozšíření samotného jazyka Ruby. Dnešním dnem je k dispozici přes 12000 rozšíření[16]. Tento nástroj dává vývojáři možnost automatizace procesu stažení a instalace pluginů pomocí několika jednoduchých příkazů. Instalace balíčku se provede příkazem `gem install <název balíčku>`. V při dlouhodobějším vývoji oceníme i možnost aktualizace doplňku. Aktualizovat můžeme příkazem `gem update <název balíčku>`.

Balíčkovací systém RubyGems však není jedinou možností, jak nainstalovat či použít doplňky třetích stran. Přímo ve frameworku Ruby on Rails je přítomen skript pro možnost instalovat doplňky alternativním způsobem pomocí příkazu `ruby script/plugin install <zdroj>`. Jako zdroj je možné uvést například adresu SVN či GIT repozitáře.

Seznam pluginů a doplňků použitých při implementaci IS Planner dostupný v příloze A obsahující implementaci IS.

6.4 Použité návrhové vzory

Návrhové vzory představují obecné řešení problému, které se využívá při návrhu software. Nejpoužívanější návrhový vzor Active Record byl již popsán v kapitole 6.3.2.1. Nyní popíšu další návrhové vzory použité při implementaci informačního systému a na jednoduchých ukázkách předvedu jejich implementaci v jazyku Ruby nebo přímo frameworku Ruby on Rails.

6.4.1 Singleton

Vzor Singleton slouží k zajištění pouze jedné instance dané třídy [7].

Možností jak implementovat takovéto chování v Ruby je spoustu. Standardní knihovna Ruby však obsahuje modul Singleton, který implementuje potřebné chování vzoru Singleton. Následující výpis kódu má demonstrovat příklad jeho použití. Na řádku 4 příkaz `include`, který pracuje s moduly, vloží modul do definice třídy `Logger`. Tímto jednoduchým mechanismem, který se nazývá mixin (česky označováno jako mixování), lze

[15] http://guides.rubyonrails.org/security.html
[16] Dle údajů statistik RubyGems hostingu, http://rubygems.org, duben 2010

dosáhnout vícenásobné dědičnosti[17]. Tím jsou splněny všechny předpoklady pro to, aby se třída `Logger` chovala tak, jak definuje vzor Singleton. K instanci lze pak přistupovat způsobem, který je demonstrován na řádku 8. Volání konstruktoru třídy pro vytvoření nového objektu na řádku 7 vyvolá výjimku neexistující metody. Podobným způsobem jsou využívány třídy pro logování právě ve frameworku Ruby on Rails.

```
1  require 'singleton'
2
3  class Logger
4      include Singleton
5  end
6
7  Logger.new
8  Logger.instance
```

Výpis 3: Příklad použití vzoru Singleton v jazyce Ruby

6.4.2 Iterator

Návrhový vzor Iterator řeší problém pohybu mezi prvky kolekce, které jsou sekvenčně uspořádány, bez znalosti implementace jednotlivých prvků posloupnosti [6].

Ruby implementuje iterátory pro procházení kolekcí poněkud jiným způsobem, než jazyky Java nebo C#. Iterátor je v Ruby metodou vestavěných kolekcí, která předává postupně do bloku všechny prvky kolekce. Sám o sobě nemá blok velký smysl, lze ho však předat jako parametr metodě, která ho může opakovaně volat. Tím, že metoda iterátoru přijímá blok, dovoluje Ruby nejen provádět iterování nad prvky kolekce, ale provádět i některé další operace jako aplikování libovolné funkce na všechny prvky kolekce, filtrování a redukování kolekce.

V následujícím výpisu jsou všechny ukázky použití iterátorů ekvivalentní. Jejich výstupem je vždy vytisknutí posloupnosti číslic 0 až 4.

```
1   [0,1,2,3,4]. each { |n| puts n }
2
3   for n in 0...5
4       puts n
5   end
6
7   5.times { |n| puts n }
```

Výpis 4: Příklad použití vzoru Iterator v jazyce Ruby

[17]Jazyk Ruby nepodporuje vícenásobnou dědičnost jako takovou (třída nemůže mít více předků), ale pomocí mixinů nabízí všechny výhody, které vícenásobné dědění poskytuje.

6.4.3 Factory

Návrhový vzor Factory je takzvaně tvořivý objektový vzor. Řeší problém, jakým způsobem rozhodnout, až v průběhu programu, o vytvoření instance konkrétní třídy [6]. Z tohoto zařazení vyplývá, že je určen pro vytváření instancí.

Ve frameworku Ruby on Rails má své místo právě například u výše zmíněné knihovny pro objektově-relační mapování. Naváži-li na předchozí výpis kódu 1, pak třída `Tag` disponuje statickou metodou `create` pro vytvoření objektu a inicializaci jeho atributů zadanými hodnotami. Splňuje tedy požadavky chování, které definuje vzor Factory.

```
1  tag = Tag.create(:name => 'ruby')
2  puts tag.name # 'ruby'
```

Výpis 5: Příklad použití vzoru Factory ve frameworku Ruby on Rails

6.4.4 Observer

Vzor Observer definuje závislosti jednoho objektu k více objektům. Umožňuje šíření události, která nastala v jednom objektu, ke všem závislým objektům [6]. Opět stejně jako u vzoru Singleton je způsobů implementace více. Nejjednodušším řešením je využít modul Observer, který nabízí standardní knihovna Ruby. Tento modul implementuje potřebné chování vzoru.

Jako příklad pro demonstraci implementace vzoru se nabízí implementace funkčnosti *zaslání notifikačního e-mailu po vytvoření úkolu nebo projektu*[18] z analýzy funkčních požadavků v kapitole 5.2.

```
1  class Task < ActiveRecord::Base
2      belongs_to :user
3  end
4
5  class Project < ActiveRecord::Base
6      belongs_to :user
7      has_many  :tasks
8  end
9
10 class NotificationObserver < ActiveRecord::Observer
11     observe :task, :project
12     def after_create(record)
13         # send notification about new project or task to user (via association record.user)
14     end
15 end
```

Výpis 6: Příklad použití vzoru Observer ve frameworku Ruby on Rails

[18] Nejedná se o přesnou ukázku zdrojových kódu, které jsou použity v implementaci aplikace. Jde jen o zjednodušenou variantu pro demonstraci implementace vzoru Observer.

Pro pochopení této ukázky kódu stačí vědět, že potomci třídy `ActiveRecord::Base` disponují několika callbacky. Jedním z nich je callback `after_create`, který je spuštěn po vytvoření nového objektu přes statickou metodu `create`. Aby se předešlo duplicitě definice callbacku, jak ve třídě `Project`, tak ve třídě `Task`, můžeme vytvořit pozorovatele, který bude mít nadefinovanou společnou obsluhu tohoto callbacku pro oba tyto modely. Zaregistrování pozorovaných objektů probíhá na řádce 11 zavoláním metody `observe` ve třídě pozorovatele `NotificationObserver`.

6.4.5 Lazy loading

Lazy loading definuje způsob přístupu k datům. Výhodou tohoto přístupu je jednoduchost pro programátora, kterému odpadají starosti, zda-li má data načtena. Nevýhodou tohoto přístupu ale je, že nemusí být vždy jasné, v jaké části programu jsou data skutečně načtena. Tento přístup je vhodný především pro malý počet dotazů, například u číselníků. Při iterování nad větší kolekcí dochází k načítání dat při každé iteraci a zbytečně tak plýtváme systémovými prostředky a časem, neboť se dotazy do databáze posílají jednotlivě.

Pro příklad opět naváži na výpis kódu 6 z předchozí podkapitoly. Na prvním řádku získám z databáze instanci projektu. V tomto okamžiku nejsou v objektu `project` načtena data asociace `tasks`. Ta jsou „líně“ načtena až ve chvíli, kdy je k nim přistoupeno.

```
1  project = Project.find(1)  # SELECT * FROM "projects" WHERE ("projects"."id" = 1);
2  project.tasks              # SELECT * FROM "tasks" WHERE ("tasks".project_id = 1);
```

Výpis 7: Příklad použití vzoru Lazy loading ve frameworku Ruby on Rails

6.4.6 Eager loading

Eager loading taktéž definuje způsob přístupu k datům. Je však mnohem vhodnější pro rozsáhlejší kolekce než lazy loading. Data se zde načítají ihned při spuštění prvního příkazu. Díky tomu jsou všechna data dostupná na klientovi a při procházení kolekce již není potřeba dělat další dotazy do databáze. Při tomto přístupu získávání dat je nutno dbát zvýšené obezřetnosti, protože je možné velmi jednoduše vytvořit objemný dotaz, který vrátí enormní množství dat, které povede k zahlcení linky nebo klienta.

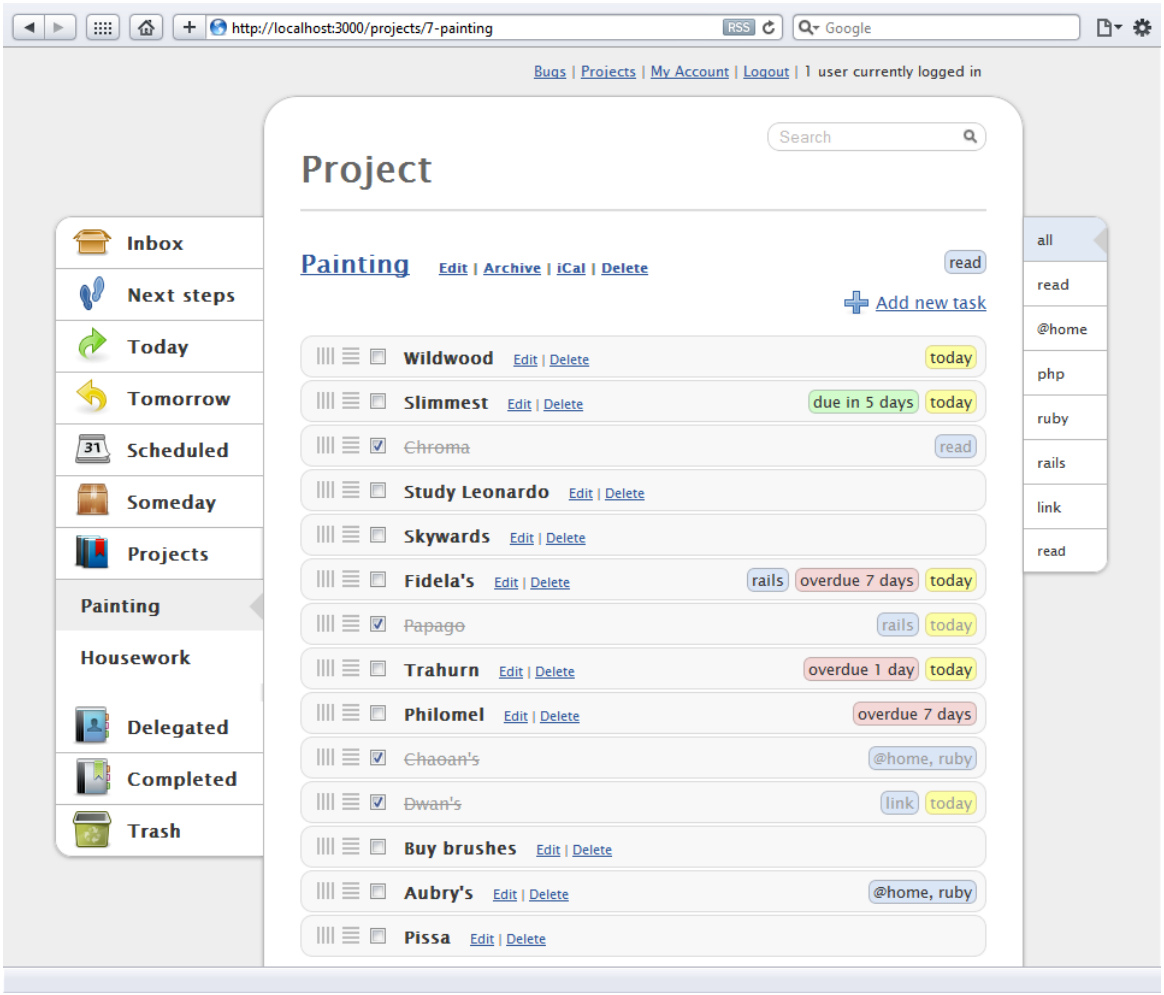
Eager loading je efektivní způsob, jak předejít problému 1+N dotazů. Pokud bychom nepoužili eager loading v příkladě 8, došlo by při každém přístupu k asociaci `tasks` k načítání úkolů pro daný projekt. Pokud by se v tomto okamžiku nacházelo v databázi 100 projektů, pak by na databázi bylo spuštěno 101 dotazů po průchodu cyklu `for`. Použitím eager loadingu jsme počet těchto dotazů snížili na pouhé dva, které se vykonají ihned na řádku 1. Při iterování cyklem `for` je kolekce `projects` včetně asociace `tasks` načtena a žádné další dotazy nebudou spuštěny.

```
1  projects = Project.find(:all, :include => 'tasks')
2  # SELECT * FROM "projects";
3  # SELECT "tasks".* FROM "tasks" WHERE ("tasks".project_id IN (1,2,3,4));
4
5  for project in Project.find(:all, :include => 'tasks')
6      puts "Project: #{project.name}, last activity : #{project.tasks.last.created_at.to_s}"
7  end
```

Výpis 8: Příklad použití vzoru Eager loading ve frameworku Ruby on Rails

6.5 Výstup implementace

Výsledkem implementace je informační systém Planner, který splňuje požadavky kladené na moderní webové aplikace popsané v kapitole 4. Aplikace je díky frameworku postavená na architektonickém vzoru Model-view-controller, poskytuje REST rozhraní pro přístup ke zdrojům a nese v sobě prvky RIA aplikace. Nasazení aplikace je navíc díky migracím a přenositelnosti databázového schématu možné na jakékoliv moderní SŘBD podporovaný modulem Active Record.



Obrázek 8: IS Planner

7 Možná budoucí rozšíření

Implementace informačního systému Planner se zcela jistě dá dále rozšiřovat a vyvíjet. Framework Ruby on Rails se ukázal jako velmi vhodný pro vývoj webových aplikací. Díky jasně definovaným principům, konvencím, adresářové struktuře a logice frameworku je i začínající programátor v tomto frameworku schopný se rychle zorientovat a ihned se zapojit do vývoje či údržby existující aplikace. K rychlému zaučení začínajícího programátora pomáhá i výborná a obsáhlá dokumentace frameworku s množstvím reálných příkladů.

Následující požadavky nespadají mezi základní, ale představují plánované a uvažované rozšíření aplikace v dalších iteracích vývoje, a mohou tak být předmětem dalších prací na informačním systému Planner.

7.1 Rozšíření o uživatelské role

Současná implementace informačního systému Planner, která je výsledkem této práce, nerozlišuje uživatelské role. Implementace je založena, po vzoru populárních moderních webových služeb, na modelu jednotného přístupu k uživatelům jako rovnocenným. Uživatelský učet se vytváří procesem uživatelské registrace, která se skládá z několika kroků a je vykonána samotným uživatelem.

1. registrace přes rozhraní registračního formuláře
2. odeslání aktivačního e-mailu na e-mailovou adresu nového uživatele
3. potvrzení registrace a aktivace účtu pomocí URL adresy s jedinečným aktivačním klíčem
4. odeslání informačního e-mailu o úspěšném dokončení registrace a aktivaci uživatele
5. přihlášení uživatele do informačního systému

Systém také obsahuje možnost znovu zadání hesla v případě jeho zapomenutí. Celý proces je podobný jako proces registrace účtu.

1. zadání nového požadavku pro reset hesla
2. odeslání ověřujícího e-mailu na e-mailovou adresu uživatele (pro ověření autencitity požadavku)
3. potvrzení požadavku pomocí URL adresy s jedinečným klíčem
4. uživatel je ověřen a přihlášen do informačního systému
5. přesměrování na stránku editace svého účtu pro možnost změny svého hesla

Výhodou tohoto modelu přístupu k vytváření uživatelských účtů je plná automatizace. Není třeba vyčlenit osobu, která bude v administraci vytvářet uživatelské přístupy „na počkání “ a přístupové údaje odesílat uživatelům elektronickou poštou. Pokud by byl ale tento model v praxi shledán jako zdlouhavý či nevyhovující, protože nenabízí kontrolu registrovaných účtů, nabízí se řešení aplikaci rozšířit o uživatelské role. Privilegovaným uživatelům by pak byla dána možnost vytvářet uživatele přímo v administraci tak, jak je popsáno na začátku tohoto odstavce, nebo schvalovat aktivace v aplikaci manuálně.

7.2 Autentizace pomocí protokolu LDAP

Současná implementace informačního systému Planner využívá autentizačního pluginu Authlogic[19]. Aplikace tak používá svůj autentizační mechanismus, jehož chování definuje implementace pluginu. Pro plugin však existuje doplněk[20], který umožní autentizaci uživatelů oproti protokolu LDAP. Odpadl by tak proces registrace uživatele a přibyla by možnost využití jednotného loginu a hesla vůči ostatním školním informačním systémům. Jako důsledek této úpravy by ale bylo vhodné z aplikace také odstranit funkci vyresetování zapomenutého hesla, jelikož by dále pozbývala významu.

7.3 Internacionalizace a lokalizace aplikace

Problém internacionalizace (i18n) a lokalizace (l10n) aplikace (bývá občas znám pod zkratkou NLS - Native Language Support) má za úkol přinést řešení podpory přirozeného jazyka, kterým aplikace komunikuje s uživatelem.

Výsledkem internacionalizace je podpora více jazyků ze strany aplikace. Lokalizace zase zahrnuje podporu mezinárodních standardů pro formáty data, času, měny atd. v již internacionalizované aplikaci.

Framework Ruby on Rails obsahují podporu pro jednoduchou internacionalizaci a lokalizaci aplikace pomocí modulu I18n. Použití je opravdu velmi jednoduché.

```
1  I18n.backend.store_translations :cz, :written_by => 'Napsal: {{name}}!'
2
3  I18n.locale = :cz
4  I18n.translate :written_by, :name => 'Roman'
5  I18n.localize Time.now
```

Výpis 9: Příklad použití internacionalizace a lokalizace pomocí modulu I18n

Ve frameworku jde podpora samozřejmě ještě dále. Framework poskytuje například helpery pro snažší použití v šablonách, překlad chybových hlášek validací nebo automatické načítání souborů s lokalizací ve formátu yaml.

Před samotnou lokalizací a internacionalizací aplikace je třeba si položit otázku: Je opravdu nutné či vhodné řešit podporu přirozeného jazyka? Cílová skupina uživatelů

[19] URL repozitáře pluginu Authlogic na serveru Github.com: http://github.com/binarylogic/authlogic

[20] URL repozitáře pluginu Authlogic LDAP na serveru Github.com: http://github.com/binarylogic/authlogic_ldap

informačního systému Planner, tým skupiny Virtlab, jsou vysokoškolsky vzdělaní lidé se znalostí anglického jazyka, kteří se v prostředí právě anglického jazyka často denně při své práci setkávají. Terminologie metodiky Getting Things Done se ujala i v našich končinách v původním jazyce publikace a autora metodiky. Je tedy zcela běžné v tomto prostředí používat původní anglická označení jako seznam *Inbox, Next Actions* či *Today*. Krok k lokalizaci a internacionalizaci aplikace by tak mohl být kontraproduktivním a je potřeba jej opravdu zvážit.

8 Závěr

V této bakalářské práci jsem čtenáře uvedl do problematiky projektového řízení, seznámil jej s jednou ze současně populárních metodik organizace práce a s vlastnostmi a požadavky, které jsou kladeny na moderní webové aplikace.

Výstupem této práce je analýza, návrh a částečná implementace informačního systému Planner. Implementovaná aplikace rozhodně není finální řešení, nic však nebrání v jejím používání. Implementuje podstatnou funkčnost a poměrně složitou logiku metodiky GTD a dále správu projektů a úkolů, včetně jejich delegace mezi uživateli. Další podrobnější požadavky mohou být implementovány dle potřeb, některé návrhy na další rozšíření jsou zmíněny v závěru práce.

Dále jsem také popsal aplikační framework Ruby on Rails, ve kterém byl informační systém implementován jako webová aplikace, a jehož ovládnutí bylo pro mě společně s ovládnutím metodiky GTD největším přínosem této práce.

Teoretický rozbor a analýza projektového řízení v úvodu práce pak shrnuje obecná fakta, která by mohla pomoci týmu Virtlab k efektivnějšímu vývoji a organizaci projektů, nehledě na to, zda-li bude k těmto účelům informační systém Planner použit. Nejpřínosnější částí této kapitoly je, dle mého osobního názoru, podkapitola týkající se týmových rolí. Díky vědomosti na jaké práce v týmu se jednotliví členové hodí a na jaké ne, se mohou lépe začlenit a vybrat si práci, která více vyhovuje jejich vlastnostem. Zjištění svých týmových rolí a kompetencí pro týmovou spolupráci, je pro mě jedním z dalších významných přínosů této práce.

Dalším podstatným tématem bylo seznámení se s problematikou metody organizace práce Getting Things Done. Metodiku jsem se pokusil zavést do svého života a po přibližně půl roce používání ji mohu zhodnotit kladně. Tím, že si uvědomuji důležité myšlenky této metodiky, jednotlivé kroky projektového plánování a řízení pracovního procesu, dokážu lépe organizovat svou práci a lépe soustředit své snažení na úkoly, které se denně objevují. Pozoruji však u sebe, že účinnost metodiky klesá s menším počtem úkolů, které jsou na obzoru. Metodika je především cílená na jednotlivce, než na celé týmy. Odhaduji, že bude ještě efektivní na malých týmech, přibližně do tuctu členů. Jde pouze o osobní odhad, proto by bylo zajímavé vidět, jestli by se jednotlivci větších týmů dokázali adaptovat na filozofii této metodiky a zda-li by vyhovovala jejich potřebám. Troufám si tvrdit, že pokud by se metodika implementovala jako doplněk k jejich zavedeným firemním procesům a kultuře, mohla by být úspěšná. Metodika GTD může člověku pomoci nejen v oblasti řízení projektů, ale pokud ji přijme jako filozofii, tak i v dalších odvětvích práce či dokonce v osobním životě.

9 Literatura

[1] Allen, D.: *Mít vše hotovo: Jak zvládnout práci i život a cítit se při tom dobře*. Jan Melvil Publishing, 2008, iSBN 978-80-903912-8-4.

[2] Belbin, M.: *Team Roles at Work*. A Butterworth-Heinemann Title, 1996, iSBN 978-0750626750.

[3] Bernard, B.: Úvod do architektury MVC. [online], 2009-05-07 [cit. 2010-01-22], iSSN 1803-5620.
URL http://zdrojak.root.cz/clanky/uvod-do-architektury-mvc/

[4] Bernard, B.: Prezentační vzory z rodiny MVC. [online], 2009-05-11 [cit. 2010-01-21], iSSN 1803-5620.
URL
http://zdrojak.root.cz/clanky/prezentacni-vzory-zrodiny-mvc/

[5] Bělohlávek, F.: *Jak vést svůj tým*. Grada Publishing, a.s., 2008, iSBN 978-80-247-1975-7.

[6] Dvořák, M.: *Návrhové vzory (design paterns)*. Diplomová práce, Vysoká škola ekonomická, 2003.

[7] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Publishing Company, 2003, iSBN 978-0321127426.

[8] Holas, T.: Programujeme v RoR. [online], 2008 [cit. 2010-04-27].
URL http://www.iquest.cz/ruby-on-rails

[9] Knesl, J.: Agilní vývoj: Úvod. [online], 2009-12-11 [cit. 2010-04-25], iSSN 1803-5620.
URL http://zdrojak.root.cz/clanky/agilni-vyvoj-uvod/

[10] Malý, M.: REST: architektura pro webové API. [online], 2009-08-03 [cit. 2010-04-21], iSSN 1803-5620.
URL http:
//zdrojak.root.cz/clanky/rest-architektura-pro-webove-api/

[11] Mrázek, P.: IT podpora řízení projektů. [online], 2007 [cit. 2010-04-17].
URL http://www.wilytech.cz/upload/it_system4_002.pdf

[12] Peter, R.: Týmová práce I. [online], 2007-06-09 [cit. 2010-04-10], iSSN 1802-6206.
URL http://www.inovace.cz/for-business/manazerske-dovednosti/clanek/tymova-prace/

[13] Peter, R.: Týmová práce II. - metody používané pro řešení projektů. [online], 2007-06-09 [cit. 2010-04-10], iSSN 1802-6206.
URL http:
//www.inovace.cz/for-business/manazerske-dovednosti/clanek/tymova-prace-ii----metody-pouzivane-pro-reseni-projektu/

[14] Pichlík, R.: Rich Internet Application. [online], 2005-06-14 [cit. 2010-04-05].
URL http://interval.cz/clanky/rich-internet-application/

[15] Roubíček, A.: Active Record vs. Repository pattern. [online], 2008-05-21 [cit. 2010-04-22].
URL http://rarous.net/weblog/271-active-record-vs-repository-pattern.aspx

[16] Ruby, S.; Thomas, D.; Hansson, D. H.: *Agile Web Development with Rails, Third Edition*. Pragmatic Bookshelf, 2009, iSBN 978-1934356166.

[17] Tichý, J.: *Programová podpora tvorby webových aplikací*. Diplomová práce, Vysoká škola ekonomická, 2004.

[18] Vlach, M.: Projektové řízení. [online], 2007-01-17 [cit. 2010-04-24].
URL http://navolnenoze.cz/blog/projektove-rizeni/

[19] Zbiejczuk, A.: *Web 2.0-charakteristika a služby*. Diplomová práce, Masarykova Univerzita, 2007.

A Implementace informačního systému

Na přiloženém kompaktním disku je umístěna elektronická podoba této bakalářské práce, a to ve formátu PDF. V adresáři `planner` se nachází implementovaný informační systém Planner postavený nad frameworkem Ruby on Rails.

A.1 Součásti aplikace

Grafika realizovaná CSS styly je důsledně oddělaná od obsahu a jeho logické struktury (princip oddělení formy od obsahu), což umožňuje vytvářet pro tytéž stránky zcela různé podoby.

A.2 Použité pluginy a gemy

Framework Ruby on Rails díky svému celosvětovému rozšíření, otevřenosti a popularitě u komunity vývojářů obsahuje velké množství různorodých pluginů rozšiřující funkčnost jazyka či komponenty implementující nějakou ucelenou funkčnost.

- authlogic, verze 2.1.3
- formtastic, verze 0.9.7
- acts-as-taggable-on
- icalendar, verze 1.1.3
- acts_as_state_machine
- acts_as_list

A.3 Instalace

Ke zprovoznění implementovaného informačního systému není potřeba k dispozici žádný webový server.

Aplikace rovněž není přímo závislá na konkrétním SŘBD. Výchozím relačním databázovým systémem byl zvolen SŘBD SQLite3 pro jeho jednoduchou a nenáročnou instalaci, rozšířenost na mnoha platformách a přenositelnost. Aplikace byla také testována na SŘBD Oracle 11g a PostgreSQL 8.3. Jediným požadavkem je přítomnost programovacího jazyka Ruby a balíčkovacího systému RubyGems v operačním systému.

Instalace a nastavení aplikace se provede následovně:

1. Spusťte příkazový řádek ve vašem operačním systému
2. Příkazem `gem install rails --include-dependencies --version=2.3.5` nainstalujte framework Ruby on Rails
3. Na příkazovém řádku se přepněte do adresáře s aplikací

4. Příkazem `rake gems:install` nainstalujte potřebné gemy a pluginy
5. Příkazem `rake setup` vytvořte databáze, databázové schéma a načtěte testovací data do databáze
6. Spusťte aplikační server příkazem `ruby script/server`
7. Otevřete aplikaci v prohlížeči na adrese `http://localhost:3000/`

Ve výchozí konfiguraci jsou zaregistrováni dva uživatelé (john, penny). Každý z uživatelů má přístup jen ke svým úkolům, projektům a do svého osobního nastavení. Oba uživatelé mají nastaveno heslo demo.

B Programátorská dokumentace

Na přiloženém kompaktním disku je umístěna elektronická podoba programátorské dokumentace ve formátu PDF. Nachází se v souboru `programatorska_prirucka.pdf`. Programátorská dokumentace obsahuje popis adresářové struktury aplikace, datový slovník a „kuchařku“ postupů pro základní programátorské práce ve frameworku Ruby on Rails, jako jsou přidání atributu, tabulky či modelu do aplikace.

C Uživatelská dokumentace

Na přiloženém kompaktním disku je umístěna elektronická podoba uživatelské dokumentace ve formátu PDF. Nachází se v souboru `uzivatelska_prirucka.pdf`. Uživatelská dokumentace obsahuje popis použití a doporučené postupy pro práci se systémem.